Manual de Referência do PostgreSQL 7.2

The PostgreSQL Global Development Group

Manual de Referência do PostgreSQL 7.2

by The PostgreSQL Global Development Group

Legal Notice

Table of Contents

Prefácio

As entradas deste *Manual de Referência* se propõem a fornecer um resumo completo e formal dos respectivos assuntos. Mais informações sobre a utilização do PostgreSQL, sob forma narrativa, de tutorial, ou de exemplos, podem ser encontradas em outras partes do conjunto de documentos do PostgreSQL. Veja as referências cruzadas listadas em cada página de referência.

As informações contidas no *Manual de Referência* também estão disponíveis no formato “man pages” do Unix.

Projeto PostgreSQL Br[1] - O centro de informações para os usuários brasileiros.

- Traduzido por Halley Pacheco de Oliveira[2] - Câmara Municipal do Rio de Janeiro.
- Revisado por Diogo de Oliveira Biazus[3] - Ikono

1. http://pgsqlbr.querencialivre.rs.gov.br
2. mailto:halleypo@yahoo.com.br
3. mailto:diogo@ikono.com.br

I. Comandos SQL

Esta parte contém informações de referência para os comandos SQL suportados pelo PostgreSQL. Por "SQL" entenda-se a linguagem SQL de modo geral; informações sobre a conformidade e a compatibilidade de cada comando com relação ao padrão podem ser encontradas nas respectivas páginas de referência.

ABORT

Name

`ABORT` — aborta a transação corrente

Synopsis

```
ABORT [ WORK | TRANSACTION ]
```

Entradas

Nenhuma.

Saídas

`ROLLBACK`

Mensagem retornada se o comando for executado com sucesso.

`NOTICE: ROLLBACK: no transaction in progress`

Se não houver nenhuma transação sendo executada.

Descrição

O comando `ABORT` desfaz a transação corrente, fazendo com que todas as modificações realizadas pela transação sejam rejeitadas. Este comando possui um comportamento idêntico ao do comando `ROLLBACK` do SQL92, estando presente apenas por razões históricas.

Notas

Use o comando `COMMIT` para terminar uma transação com sucesso.

Utilização

Para abortar todas as modificações:

```
ABORT WORK;
```

Compatibilidade

SQL92

Este comando é uma extensão do PostgreSQL presente apenas por razões históricas. O comando equivalente do SQL92 é o `ROLLBACK`.

ALTER GROUP

Name

`ALTER GROUP` — inclui ou exclui usuários em um grupo

Synopsis

```
ALTER GROUP nome ADD USER nome_do_usuário [, ... ]
ALTER GROUP nome DROP USER nome_do_usuário [, ... ]
```

Entradas

nome

O nome do grupo a ser modificado.

nome_do_usuário

Os usuários a serem incluídos ou excluídos no grupo. Os nomes dos usuários devem existir.

Saídas

`ALTER GROUP`

Mensagem retornada se a alteração for realizada com sucesso.

Descrição

O comando `ALTER GROUP` é utilizado para incluir ou excluir usuários em um grupo. Somente os superusuários do banco de dados podem utilizar este comando. Incluir um usuário em um grupo não cria o usuário. Analogamente, excluir um usuário de um grupo não exclui o usuário do banco de dados.

Use o *CREATE GROUP* para criar um novo grupo, e o *DROP GROUP* para remover um grupo.

Utilização

Incluir usuários em um grupo:

```
ALTER GROUP arquitetura ADD USER joana, alberto;
```

Excluir um usuário de um grupo:

```
ALTER GROUP engenharia DROP USER margarida;
```

Compatibilidade

SQL92

Não existe o comando `ALTER GROUP` no SQL92. O conceito de “roles” é similar ao de grupos.

ALTER TABLE

Name

`ALTER TABLE` — altera a definição da tabela

Synopsis

```
ALTER TABLE [ ONLY ] tabela [ * ]
    ADD [ COLUMN ] coluna tipo [ restrição_de_coluna [ ... ] ]
ALTER TABLE [ ONLY ] tabela [ * ]
    ALTER [ COLUMN ] coluna { SET DEFAULT valor | DROP DEFAULT }
ALTER TABLE [ ONLY ] tabela [ * ]
    ALTER [ COLUMN ] coluna SET STATISTICS inteiro
ALTER TABLE [ ONLY ] tabela [ * ]
    RENAME [ COLUMN ] coluna TO novo_nome_da_coluna
ALTER TABLE tabela
    RENAME TO novo_nome_da_tabela
ALTER TABLE tabela
    ADD definição_de_restrição_de_tabela
ALTER TABLE [ ONLY ] tabela
 DROP CONSTRAINT restrição { RESTRICT | CASCADE }
ALTER TABLE tabela
 OWNER TO novo_dono
```

Entradas

tabela

O nome da tabela existente a ser alterada.

coluna

O nome de uma coluna nova ou existente.

tipo

O tipo da nova coluna.

novo_nome_da_coluna

O novo nome para a coluna existente.

novo_nome_da_tabela

O novo nome para a tabela.

definição_de_restrição_de_tabela

A nova restrição de tabela (table constraint) para a tabela.

novo_dono

O nome de usuário do novo dono da tabela.

Saídas

ALTER

Mensagem retornada se o nome da coluna ou da tabela for alterado com sucesso.

ERROR

Mensagem retornada se a tabela ou a coluna não existir.

Descrição

O comando ALTER TABLE altera a definição de uma tabela existente. A forma ADD COLUMN adiciona uma nova coluna na tabela utilizando a mesma sintaxe de *CREATE TABLE*. A forma ALTER COLUMN SET/DROP DEFAULT permite definir ou remover o valor padrão para a coluna. Note que o valor padrão somente se aplica aos próximos comandos INSERT; as linhas existentes na tabela não são modificadas. A forma ALTER COLUMN SET STATISTICS permite controlar a coleta de estatísticas para as operações *ANALYZE* posteriores. A cláusula RENAME faz com que o nome da tabela, coluna, índice ou seqüência seja mudado sem que os dados sejam modificados. Os dados permanecem do mesmo tipo e tamanho após o comando ser executado. A cláusula ADD *definição_de_restrição_de_tabela* adiciona uma nova restrição de tabela utilizando a mesma sintaxe de *CREATE TABLE*. A cláusula DROP CONSTRAINT *restrição* elimina todas as restrições da tabela (e de suas descendentes) que correspondam à *restrição*. A cláusula OWNER muda o dono da tabela para o usuário *novo_dono*.

Somente o dono da tabela pode modificar seu esquema.

Notas

A palavra chave COLUMN é informativa podendo ser omitida.

Na atual implementação de ADD COLUMN, as cláusulas valor padrão e NOT NULL não são suportadas para a nova coluna. Pode ser usada a forma SET DEFAULT do comando ALTER TABLE para definir o valor padrão mais tarde. (Os valores atuais das linhas existentes poderão ser atualizados, posteriormente, para o novo valor padrão usando o comando *UPDATE*.)

Em DROP CONSTRAINT a palavra chave RESTRICT é requerida, embora as dependências ainda não sejam verificadas. A opção CASCADE não é suportada. Atualmente DROP CONSTRAINT remove somente as restrições CHECK. Para remover as restrições PRIMARY ou UNIQUE deve ser removido o índice correspondente utilizando o comando *DROP INDEX*. Para remover uma FOREIGN KEY é necessário recriar e recarregar a tabela usando outros parâmetros no comando *CREATE TABLE*.

Por exemplo, para remover todas as restrições da tabela `distribuidores`:

```
CREATE TABLE temp AS SELECT * FROM distribuidores;
DROP TABLE distribuidores;
CREATE TABLE distribuidores AS SELECT * FROM temp;
DROP TABLE temp;
```

Somente o dono da tabela pode alterá-la. Mudar qualquer parte do esquema do catálogo do sistema não é permitido. O manual *Guia do Usuário do PostgreSQL* possui mais informações sobre herança.

Consulte o comando `CREATE TABLE` para obter uma descrição mais detalhada dos argumentos válidos.

Utilização

Para adicionar uma coluna do tipo `VARCHAR` à tabela:

```
ALTER TABLE distribuidores ADD COLUMN endereco VARCHAR(30);
```

Para mudar o nome de uma coluna existente:

```
ALTER TABLE distribuidores RENAME COLUMN endereco TO cidade;
```

Para mudar o nome de uma tabela existente:

```
ALTER TABLE distribuidores RENAME TO fornecedores;
```

Para adicionar uma restrição de verificação (CHECK) a uma tabela:

```
ALTER TABLE distribuidores ADD CONSTRAINT cep_chk CHECK (char_length(cod_cep) = 8);
```

Para remover uma restrição de verificação de uma tabela e de todas as suas filhas:

```
ALTER TABLE distribuidores DROP CONSTRAINT cepchk RESTRICT;
```

Para adicionar uma chave estrangeira a uma tabela:

```
ALTER TABLE distribuidores ADD CONSTRAINT fk_dist FOREIGN KEY (endereco) REFERENCES e
```

Para adicionar uma restrição de unicidade (multi-coluna) à tabela:

```
ALTER TABLE distribuidores ADD CONSTRAINT dist_id_cep_key UNIQUE (dist_id, cep);
```

Para adicionar uma restrição de chave primária a uma tabela com o nome gerado automaticamente, observando-se que a tabela somente pode possuir uma única chave primária:

```
ALTER TABLE distribuidores ADD PRIMARY KEY (dist_id);
```

Compatibilidade

SQL92

A forma `ADD COLUMN` está em conformidade, a não ser por não suportar valor padrão e NOT NULL, conforme foi explicado anteriormente. A forma `ALTER COLUMN` está em conformidade total.

O SQL92 especifica algumas funcionalidades adicionais para o comando `ALTER TABLE` que ainda não são diretamente suportadas pelo PostgreSQL:

```
ALTER TABLE tabela DROP [ COLUMN ] coluna { RESTRICT | CASCADE }
```

Remove a coluna da tabela. Na implementação atual, para remover uma coluna existente a tabela deve ser recriada e recarregada:

```
CREATE TABLE temp AS SELECT did, cidade FROM distribuidores;
DROP TABLE distribuidores;
CREATE TABLE distribuidores (
    did        DECIMAL(3)  DEFAULT 1,
    cidade     VARCHAR(40) NOT NULL
);
INSERT INTO distribuidores SELECT * FROM temp;
DROP TABLE temp;
```

As cláusulas para mudar o nome das tabelas, colunas, índices e seqüências são extensões do PostgreSQL ao SQL92.

ALTER USER

Name

ALTER USER — altera a conta de um usuário do banco de dados

Synopsis

```
ALTER USER nome_do_usuário [ [ WITH ] opção [ ... ] ]

onde opção pode ser:

   [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'senha'
        | CREATEDB | NOCREATEDB
        | CREATEUSER | NOCREATEUSER
        | VALID UNTIL 'data_hora'
```

Entradas

nome_do_usuário

O nome do usuário cuja conta está sendo alterada.

senha

A nova senha a ser utilizada para esta conta.

ENCRYPTED
UNENCRYPTED

Estas palavras chave controlam se a senha é armazenada criptografada, ou não, em `pg_shadow` (Consulte o comando *CREATE USER* para obter mais informações sobre esta opção).

CREATEDB
NOCREATEDB

Estas cláusulas definem a permissão para o usuário criar bancos de dados. Se CREATEDB for especificado, o usuário sendo alterado terá permissão para criar seus próprios bancos de dados. Especificando-se NOCREATEDB, a permissão para criar bancos de dados é negada ao usuário.

CREATEUSER
NOCREATEUSER

Estas cláusulas definem a permissão para o usuário criar novos usuários. Esta opção também torna o usuário um superusuário, que pode mudar todas as restrições de acesso.

data_hora

A data (e, opcionalmente, a hora) de expiração da senha do usuário.

Saídas

`ALTER USER`

Mensagem retornada se a alteração for realizada com sucesso.

`ERROR: ALTER USER: user "nome_do_usuário" does not exist`

Mensagem retornada quando o usuário especificado não existir no banco de dados.

Descrição

O comando `ALTER USER` é utilizado para mudar os atributos da conta de um usuário do PostgreSQL. Os atributos não mencionados no comando permanecem com os seus valores inalterados.

Somente um superusuário do banco de dados pode alterar os privilégios e a expiração da senha com este comando. Os usuários comuns somente podem alterar as suas próprias senhas.

O comando `ALTER USER` não pode mudar a participação do usuário nos grupos. Use o *ALTER GROUP* para realizar esta operação.

Use o *CREATE USER* para criar um novo usuário, e o *DROP USER* para remover um usuário.

Utilização

Mudar a senha do usuário:

```
ALTER USER marcos WITH PASSWORD ’hu8jmn3’;
```

Mudar a data de expiração da senha do usuário:

```
ALTER USER manuel VALID UNTIL ’Jan 31 2030’;
```

Mudar a data de expiração da senha do usuário, especificando que sua autorização expira ao meio dia de 4 de maio de 1998, usando uma zona horária uma hora adiante da UTC:

```
ALTER USER cristina VALID UNTIL ’May 4 12:00:00 1998 +1’;
```

Dar ao usuário poderes para criar outros usuários e novos bancos de dados:

```
ALTER USER marcela CREATEUSER CREATEDB;
```

Compatibilidade

SQL92

Não existe o comando `ALTER USER` no SQL92. O padrão deixa a definição dos usuários para a implementação.

ANALYZE

Name

`ANALYZE` — coleta estatísticas sobre um banco de dados

Synopsis

```
ANALYZE [ VERBOSE ] [ tabela [ (coluna [, ...] ) ] ]
```

Entradas

VERBOSE

Ativa a exibição de mensagens de progresso.

`tabela`

O nome de uma tabela específica a ser analisada. Por padrão todas as tabelas.

`coluna`

O nome de uma coluna específica a ser analisada. Por padrão todas as colunas.

Saídas

`ANALYZE`

O comando está concluído.

Descrição

O comando `ANALYZE` coleta estatísticas sobre o conteúdo das tabelas do PostgreSQL, e armazena os resultados na tabela do sistema `pg_statistic`. Posteriormente, o otimizador de consultas utiliza estas estatísticas para auxiliar na determinação do plano de execução mais eficiente para as consultas.

Sem nenhum parâmetro, o comando `ANALYZE` analisa todas as tabelas do banco de dados em uso. Com parâmetro, o comando `ANALYZE` analisa somente uma tabela. É possível ainda fornecer uma relação de nomes de colunas e, neste caso, somente as estatísticas para estas colunas são atualizadas.

Notas

Aconselha-se executar o comando `ANALYZE` periodicamente, ou logo após realizar uma alteração significativa no conteúdo de uma tabela. Estatísticas precisas auxiliam o otimizador na escolha do plano de consulta mais apropriado e, portanto, a melhorar o tempo do processamento da consulta. Uma estratégia habitual é executar *VACUUM* e `ANALYZE` uma vez por dia em períodos de pouca carga.

Ao contrário do comando `VACUUM FULL`, o comando `ANALYZE` requer somente um bloqueio de leitura na tabela podendo, portanto, ser executado em conjunto com outras atividades na tabela.

Para tabelas grandes, o comando `ANALYZE` pega amostras aleatórias do conteúdo da tabela, em vez de examinar todas as linhas. Esta estratégia permite que mesmo tabelas muito grandes sejam analisadas em curto espaço de tempo. Observe, entretanto, que as estatísticas são apenas aproximadas e vão ser um pouco diferentes cada vez que o comando `ANALYZE` for executado, mesmo que o conteúdo da tabela não se altere, podendo ocasionar pequenas mudanças no custo estimado pelo otimizador mostrado no comando `EXPLAIN`.

As estatísticas coletadas geralmente incluem a relação dos valores com maior incidência de cada coluna e um histograma mostrando a distribuição aproximada dos dados de cada coluna. Um, ou ambos, podem ser omitidos se o comando `ANALYZE` considerá-los irrelevantes (por exemplo, em uma coluna com chave única não existem valores repetidos) ou se o tipo de dado da coluna não suportar os operadores apropriados. Existem mais informações sobre as estatísticas no *Guia do Usuário*.

A extensão da análise pode ser controlada ajustando-se as estatísticas por coluna através do comando `ALTER TABLE ALTER COLUMN SET STATISTICS` (consulte o comando *ALTER TABLE*). O valor especificado define o número máximo de entradas da lista de valores com maior incidência e o número máximo de barras no histograma. O valor padrão é 10, mas pode ser ajustado para mais, ou para menos, para balancear a precisão das estimativas do otimizador contra o tempo dispendido para a execução do comando `ANALYZE` e a quantidade de espaço ocupado pela tabela `pg_statistic`. Em particular, especificando-se o valor zero desativa a coleta de estatísticas para a coluna, podendo ser útil para colunas que nunca são usadas como parte das cláusulas WHERE, GROUP BY ou ORDER BY das consultas, porque as estatísticas destas colunas não têm utilidade para o otimizador.

O maior número de estatísticas, entre as colunas sendo analisadas, determina o número de linhas amostradas para preparar as estatísticas. Aumentando o número de estatísticas por coluna causa um aumento proporcional no tempo e espaço necessário para executar o comando `ANALYZE`.

Compatibilidade

SQL92

Não existe o comando `ANALYZE` no SQL92.

BEGIN

Name

`BEGIN` — inicia um bloco de transação

Synopsis

```
BEGIN [ WORK | TRANSACTION ]
```

Entradas

WORK
TRANSACTION

Palavras chave opcionais. Não produzem nenhum efeito.

Saídas

`BEGIN`

Significa que uma nova transação começou.

`NOTICE: BEGIN: already a transaction in progress`

Indica que uma transação está sendo executada. A transação corrente não é afetada.

Descrição

Por padrão, o PostgreSQL executa as transações em *modo não encadeado* (também conhecido por "autocommit" [auto-efetivação] em outros sistemas de banco de dados). Em outras palavras, cada comando é executado em sua própria transação e uma efetivação é implicitamente realizada ao final do comando (se a execução terminar com sucesso, senão um "rollback" é realizado). O comando `BEGIN` inicia uma transação no modo encadeado, ou seja, todas as declarações após o comando `BEGIN` são executadas como sendo uma única transação, até que seja encontrado um comando explícito *COMMIT* ou *ROLLBACK*, ou a execução ser abortada. Os comandos são executados mais rápido no modo encadeado, porque cada início/efetivação de transação requer uma atividade significativa de CPU e de disco. A execução de vários comandos em uma única transação também é requerida por motivo de consistência, quando várias tabelas relacionadas são modificadas.

O nível de isolamento padrão da transação no PostgreSQL é READ COMMITTED, onde os comandos dentro de uma transação enxergam apenas as mudanças efetivadas antes da execução do comando. Por esse motivo deve-se utilizar `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE` logo após o comando `BEGIN` se for necessário um nível de isolamento mais rigoroso para a transação. No modo SERIALIZABLE os comandos enxergam apenas as modificações efetivadas antes do início de toda a transação (na verdade, antes da execução do primeiro comando DML em uma transação serializável).

Se a transação for efetivada, o PostgreSQL garante que todas as atualizações são realizadas ou, então, que nenhuma delas é realizada. As transações possuem a propriedade ACID (atomicidade, consistência, isolamento e durabilidade) padrão.

Notas

Consulte o comando *LOCK* para obter mais informações sobre o bloqueio de tabelas dentro de uma transação.

Utilize *COMMIT* ou *ROLLBACK* para terminar a transação.

Utilização

Para iniciar uma transação:

```
BEGIN WORK;
```

Compatibilidade

SQL92

O comando `BEGIN` é uma extensão do PostgreSQL à linguagem.. Não existe o comando `BEGIN` explícito no SQL92; o início de uma transação é sempre implícito, e termina pelo comando `COMMIT` ou pelo comando `ROLLBACK`.

> **Note:** Muitos sistemas de banco de dados relacionais oferecem a funcionalidade de auto-efetivação (autocommit) por conveniência.

Lembre-se que a palavra chave `BEGIN` é utilizada para uma finalidade diferente no SQL embutido. Deve-se ter muito cuidado com relação à semântica da transação ao se portar aplicativos de banco de dados.

O SQL92 requer que SERIALIZABLE seja o nível de isolamento padrão das transações.

CHECKPOINT

Name

`CHECKPOINT` — força um ponto de controle no log de transação

Synopsis

```
CHECKPOINT
```

Descrição

A gravação prévia do registro no log (Write-Ahead Logging/WAL) coloca periodicamente um ponto de controle (checkpoint) no log de transação (Para ajustar o intervalo automático do ponto de controle, consulte as opções de configuração em tempo de execução *CHECKPOINT_SEGMENTS* e *CHECKPOINT_TIMEOUT*). O comando `CHECKPOINT` força um ponto de controle imediato ao ser executado, sem aguardar o ponto de controle pré-definido.

Um ponto de controle é um ponto na seqüência do log de transação no qual todos os arquivos de dados foram atualizados para refletir a informação no log. Todos os dados são escritos no disco. Consulte o *Guia do Administrador do PostgreSQL* para obter mais informações sobre o WAL.

Somente os superusuários podem executar o comando `CHECKPOINT`. A utilização deste comando durante uma operação normal não é esperada .

Consulte também

Guia do Administrador do PostgreSQL

Compatibilidade

O comando `CHECKPOINT` é uma extensão do PostgreSQL à linguagem.

CLOSE

Name

`CLOSE` — fecha o cursor

Synopsis

```
CLOSE cursor
```

Entradas

cursor

O nome do cursor aberto a ser fechado.

Saídas

`CLOSE`

Mensagem retornada se o cursor for fechado com sucesso.

`NOTICE PerformPortalClose: portal "cursor" not found`

Esta advertência é exibida quando o *cursor* não está declarado ou se já tiver sido fechado.

Descrição

O comando `CLOSE` libera os recursos associados a um cursor aberto. Após o cursor ser fechado, não é permitida nenhuma operação posterior sobre o mesmo. O cursor deve ser fechado quando não for mais necessário.

Um fechamento implícito é executado para todos os cursores abertos quando a transação é terminada pelo comando `COMMIT` ou pelo comando `ROLLBACK`.

Notas

O PostgreSQL não possui um comando explícito `OPEN` cursor; o cursor é considerado aberto ao ser declarado. Use o comando `DECLARE` para declarar um cursor.

Utilização

Fechar o cursor `cur_emp`:

```
CLOSE cur_emp;
```

Compatibilidade

SQL92

O comando `CLOSE` é totalmente compatível com o SQL92.

CLUSTER

Name

CLUSTER — agrupa uma tabela de acordo com um índice

Synopsis

```
CLUSTER nome_do_índice ON nome_da_tabela
```

Entradas

nome_do_índice

O nome de um índice.

nome_da_tabela

O nome de uma tabela.

Saídas

`CLUSTER`

O agrupamento foi realizado com sucesso.

`ERROR: relation <numero_tabelarelação> inherits "nome_da_tabela"`

`ERROR: Relation nome_da_tabela does not exist!`

Descrição

O comando `CLUSTER` instrui o PostgreSQL para agrupar a tabela especificada por *nome_da_tabela* baseado no índice especificado por *nome_do_índice*. É necessário que o índice tenha sido criado anteriormente na tabela *nome_da_tabela*.

Quando a tabela é agrupada, ela é fisicamente reordenada baseado na informação do índice. O agrupamento é estático. Em outras palavras, assim que a tabela for atualizada as modificações não serão agrupadas. Nenhuma tentativa é feita para manter as novas instâncias ou as tuplas atualizadas agrupadas. Se for desejado, a tabela pode ser reagrupada manualmente executando-se o comando novamente.

Notas

Na realidade a tabela é copiada para uma tabela temporária ordenada pelo índice e, em seguida, renomeada para o seu nome original. Por esta razão, todas as permissões de acesso concedidas e os outros índices são perdidos quando o agrupamento é realizado.

No caso de se estar acessando uma única linha da tabela aleatoriamente, a ordem física dos dados da tabela não é importante. Entretanto, havendo uma tendência para acessar alguns dados mais do que outros, se existir um índice que agrupa estes dados haverá benefício se o comando `CLUSTER` for utilizado.

Outra situação em que o comando `CLUSTER` é útil são os casos em que se usa o índice para acessar várias linhas da tabela. Se for solicitada uma faixa de valores indexados de uma tabela, ou um único valor indexado possuindo muitas linhas que correspondam a este valor, o comando `CLUSTER` ajuda porque quando o índice identifica a página da primeira linha todas as outras linhas estarão provavelmente nesta mesma página, reduzindo o acesso ao disco e acelerando a consulta.

Existem duas maneiras de se agrupar os dados. A primeira é com o comando `CLUSTER`, que reordena a tabela original na ordem do índice especificado. Este procedimento pode ser lento para tabelas grandes porque as linhas são lidas da tabela na ordem do índice e, se a tabela não estiver ordenada, as linhas estarão em páginas aleatórias, fazendo uma página do disco ser lida para cada linha movida. O PostgreSQL possui um `cache`, mas a maioria das tabelas grandes não cabem no `cache`.

Outra maneira de agrupar a tabela é usar

```
SELECT lista_de_colunas INTO TABLE nova_tabela
     FROM nome_da_tabela ORDER BY lista_de_colunas
```

que usa o código de ordenação do PostgreSQL da cláusula ORDER BY para fazer o papel do índice, o que é muito mais rápido para dados não ordenados. Em seguida a tabela original deve ser removida, o comando `ALTER TABLE...RENAME` deve ser utilizado para mudar o nome da *nova_tabela* para o nome da tabela original, e os índices da tabela devem ser recriados. O único problema é que o OID não é preservado. Deste momento em diante o comando `CLUSTER` deverá ser rápido, porque a maior parte dos dados da tabela estará ordenada, e o índice existente é usado.

Utilização

Agrupar a relação empregados baseado no atributo salário:

```
CLUSTER emp_ind ON emp;
```

Compatibilidade

SQL92

Não existe o comando `CLUSTER` no SQL92.

COMMENT

Name

COMMENT — cria ou altera o comentário de um objeto

Synopsis

```
COMMENT ON
[
  [ DATABASE | INDEX | RULE | SEQUENCE | TABLE | TYPE | VIEW ] nome_do_objeto |
  COLUMN nome_da_tabela.nome_da_coluna |
  AGGREGATE nome_da_agregação (tipo_da_agregação) |
  FUNCTION nome_da_função (arg1, arg2, ...) |
  OPERATOR op (leftoperand_type rightoperand_type) |
  TRIGGER nome_do_gatilho ON nome_da_tabela
] IS 'texto'
```

Entradas

nome_do_objeto, nome_da_tabela, nome_da_coluna, nome_da_agregação, nome_da_função, op e nome_do_gatilho

O nome do objeto ao qual o comentário se refere.

texto

O comentário a ser adicionado.

Saídas

COMMENT

Mensagem retornada se o comando for executado com sucesso.

Descrição

O comando COMMENT armazena um comentário sobre um objeto do banco de dados. Os comentários podem ser facilmente acessados através dos comandos \dd, \d+ e \l+ do psql. Outras interfaces de

usuário podem acessar os comentários utilizando as mesmas funções nativas usadas pelo `psql`, que são: `obj_description()` e `col_description()`.

Para modificar um comentário basta executar novamente o comando `COMMENT` para o mesmo objeto. Somente um único comentário é armazenado para cada objeto. Para excluir um comentário escreva `NULL` no lugar do texto. O comentário é automaticamente excluído quando o objeto é excluído.

Deve ser observado que não existe atualmente nenhum mecanismo de segurança para os comentários: qualquer usuário conectado ao banco de dados pode ver todos os comentários dos objetos do banco de dados (embora somente um superusuário possa modificar comentários de objetos que não lhe pertencem). Portanto, não coloque informações confidenciais nos comentários.

Utilização

Adicionar um comentário à tabela `minha_tabela`:

```
COMMENT ON minha_tabela IS 'Esta tabela é minha.';
```

Alguns outros exemplos:

```
COMMENT ON DATABASE bd_desenv IS 'Banco de dados de desenvolvimento';
COMMENT ON INDEX idx_func_id IS 'Garante a unicidade do identificador do funcionário
COMMENT ON RULE upd_func IS 'Registra as atualizações dos registros dos funcionários
COMMENT ON SEQUENCE seq_func IS 'Gera a chave primária dos funcionários';
COMMENT ON TABLE tbl_func IS 'Cadastro dos funcionários';
COMMENT ON TYPE tipo_cn IS 'Suporte a números complexos';
COMMENT ON VIEW vis_dep_custo IS 'Visão dos custos dos departamentos';
COMMENT ON COLUMN tbl_func.id_func IS 'Identificador do funcionário';
COMMENT ON AGGREGATE agreg_var (double precision) IS 'Calcula a variância da amostra
COMMENT ON FUNCTION func_romano (int) IS 'Retorna o número em algarismos romanos';
COMMENT ON OPERATOR ^ (text, text) IS 'Realiza a interseção de dois textos';
COMMENT ON TRIGGER gat_func ON tbl_func IS 'Utilizado para integridade referencial';
```

Compatibilidade

SQL92

Não existe o comando `COMMENT` no SQL92.

COMMIT

Name

`COMMIT` — efetiva a transação corrente

Synopsis

```
COMMIT [ WORK | TRANSACTION ]
```

Entradas

WORK
TRANSACTION

Palavras chave opcionais. Não produzem nenhum efeito.

Saídas

`COMMIT`

Mensagem retornada se a transação for efetivada com sucesso.

`NOTICE: COMMIT: no transaction in progress`

Se não houver nenhuma transação sendo executada.

Descrição

O comando `COMMIT` efetiva a transação sendo executada. Todas as modificações efetuadas pela transação se tornam visíveis para os outros, e existe a garantia de permanecerem se uma falha ocorrer.

Notas

As palavras chave WORK e TRANSACTION são informativas podendo ser omitidas.

Use o *ROLLBACK* para desfazer a transação.

Utilização

Para tornar todas as modificações permanentes:

```
COMMIT WORK;
```

Compatibilidade

SQL92

O SQL92 somente especifica as duas formas `COMMIT` e `COMMIT WORK`. Fora isso a compatibilidade é total.

COPY

Name

COPY — copia dados entre arquivos e tabelas

Synopsis

```
COPY [ BINARY ] tabela [ WITH OIDS ]
    FROM { 'nome_do_arquivo' | stdin }
    [ [USING] DELIMITERS 'delimitador' ]
    [ WITH NULL AS 'cadeia de caracteres nula' ]
COPY [ BINARY ] tabela [ WITH OIDS ]
    TO { 'nome_do_arquivo' | stdout }
    [ [USING] DELIMITERS 'delimitador' ]
    [ WITH NULL AS 'cadeia de caracteres nula' ]
```

Entradas

BINARY

Altera o comportamento da formatação dos campos, fazendo todos os dados serem escritos ou lidos no formato binário em vez de texto. As opções DELIMITERS e WITH NULL não são pertinentes para o formato binário.

tabela

O nome de uma tabela existente.

WITH OIDS

Especifica a cópia do identificador interno do objeto (OID) para cada linha.

nome_do_arquivo

O nome do arquivo de entrada ou de saída no Unix, junto com o caminho absoluto.

`stdin`

Especifica que a entrada vem do aplicativo cliente.

`stdout`

Especifica que a saída vai para o aplicativo cliente.

delimitador

O caractere que separa os campos dentro de cada linha do arquivo.

cadeia de caracteres nula

A cadeia de caracteres que representa o valor nulo. O padrão é "`\N`" (contrabarra-N). Pode-se preferir uma cadeia de caracteres vazia, por exemplo.

Note: Durante a leitura, qualquer item de dado que corresponda a esta cadeia de caracteres é armazenado com o valor nulo, portanto deve-se ter certeza de estar usando para ler a mesma cadeia de caracteres que foi usada para escrever.

Saídas

`COPY`

A cópia terminou com sucesso.

`ERROR:` *`razão`*

A cópia falhou pela razão mostrada na mensagem de erro.

Descrição

O comando `COPY` copia dados entre tabelas do PostgreSQL e arquivos do sistema operacional. O comando `COPY TO` copia todo o conteúdo de uma tabela para um arquivo, enquanto o `COPY FROM` copia os dados de um arquivo para uma tabela (adicionando os dados aos existentes na tabela).

O comando `COPY` com um nome de arquivo instrui o servidor PostgreSQL para ler ou escrever diretamente de um arquivo. O arquivo deve ser acessível ao servidor e o nome deve ser especificado a partir do ponto de vista do servidor. Quando `stdin` ou `stdout` for especificado, os dados fluirão entre o cliente e o servidor.

Tip: Não confunda o comando `COPY` com a instrução `\copy` do psql. O `\copy` executa `COPY FROM stdin` ou `COPY TO stdout` e, então, lê/grava os dados em um arquivo acessível ao cliente psql. Portanto, a acessibilidade e os direitos de acesso dependem do cliente e não do servidor, quando o `\copy` é utilizado.

Notas

O comando `COPY` só pode ser utilizado em tabelas, não pode ser utilizado em visões.

A palavra chave BINARY força todos os dados serem escritos/lidos no formato binário em vez de texto. É um pouco mais rápido do que a cópia normal, mas o arquivo binário produzido não é portável entre máquinas com arquiteturas diferentes.

Por padrão, a cópia no formato texto usa o caractere de tabulação ("\t") como delimitador de campos. O delimitador de campo pode ser mudado para qualquer outro caractere único usando a cláusula USING

DELIMITERS. Os caracteres nos campos de dados que forem idênticos ao caractere delimitador serão precedidos por uma contrabarra.

É necessário possuir *permissão de leitura* em todas as tabelas cujos valores são lidos pelo `COPY`, e *permissão para inserir* ou *atualizar* em uma tabela onde valores são inseridos pelo `COPY`. O servidor também necessita de permissões apropriadas no Unix para todos os arquivos lidos ou escritos pelo `COPY`.

O comando `COPY TO` não invoca as regras nem atua no padrão da coluna. Invoca os gatilhos e as restrições de verificação.

O `COPY` pára de executar no primeiro erro, o que não deve acarretar problemas para o `COPY FROM`, mas a tabela de destino já vai ter recebido linhas no caso do `COPY TO`. Estas linhas não são visíveis nem acessíveis, mas ocupam espaço em disco, podendo ocasionar o desperdício de uma grande quantidade de espaço em disco se o erro ocorrer durante a cópia de uma grande quantidade de dados. Deve-se executar o comando `VACUUM` para recuperar o espaço desperdiçado.

Os arquivos declarados no comando `COPY` são lidos ou escritos diretamente pelo servidor, e não pelo aplicativo cliente. Portanto, devem residir ou serem acessíveis pela máquina servidora de banco de dados, e não pela estação cliente. Os arquivos devem ser acessíveis e poder ser lidos ou escritos pelo usuário do PostgreSQL (o ID do usuário sob o qual o servidor processa), e não pelo cliente. O `COPY` com nome de arquivo só é permitido aos superusuários do banco de dados, porque permite ler e escrever em qualquer arquivo que o servidor possua privilégio de acesso.

Tip: A instrução `\copy` do psql lê e escreve arquivos na estação cliente usando as permissões do cliente, portanto não é restrita aos superusuários.

Recomenda-se que os nomes dos arquivos usados no comando `COPY` sejam sempre especificados como um caminho absoluto, sendo exigido pelo servidor no caso do `COPY TO`, mas para `COPY FROM` existe a opção de ler um arquivo especificado pelo caminho relativo. O caminho é interpretado como sendo relativo ao diretório de trabalho do servidor (algum lugar abaixo de `$PGDATA`), e não relativo ao diretório de trabalho do cliente.

Formatos dos Arquivos

Formato Texto

Quando o comando `COPY` é utilizado sem a opção BINARY, o arquivo lido ou escrito é um arquivo de texto com uma linha para cada linha da tabela. As colunas (atributos) são separadas na linha pelo caractere delimitador. Os valores dos atributos são cadeias de caracteres geradas pela função de saída, ou aceitáveis pela função de entrada, para cada tipo de dado de atributo. A cadeia de caracteres nula especificada é utilizada no lugar dos atributos que são nulos.

Se WITH OIDS for especificado, o OID é lido ou escrito como a primeira coluna, precedendo as colunas de dado do usuário (Vai acontecer um erro se WITH OIDS for especificado para uma tabela que não possua OIDs).

O fim dos dados pode ser representado por uma única linha contendo apenas contrabarra-ponto (`\.`). Um marcador de fim de dados não é necessário ao se ler de um arquivo Unix, porque o fim de arquivo serve

perfeitamente bem; mas um marcador de fim dos dados deve ser fornecido para copiar os dados de ou para um aplicativo cliente.

O caractere contrabarra (`\`) pode ser usado nos dados do comando `COPY` para evitar que caracteres dos dados sejam interpretados como delimitadores de linha ou de coluna. Em particular, os seguintes caracteres *devem* ser precedidos por uma contrabarra se aparecerem como parte do valor de um atributo: a própria contrabarra, a nova-linha e o caractere delimitador corrente.

As seguintes seqüências especiais de contrabarra são reconhecidas pelo comando `COPY FROM`:

Seqüência	Representa
`\b`	Backspace (ASCII 8)
`\f`	Avanço de página (ASCII 12)
`\n`	Nova-linha (ASCII 10)
`\r`	Retorno do carro (ASCII 13)
`\t`	Tabulação (ASCII 9)
`\v`	Tabulação Vertical (ASCII 11)
`\`*`dígitos`*	A contrabarra seguida por um a três dígitos octais especifica o caractere com este código numérico

Atualmente o `COPY TO` nunca gera uma seqüência contrabarra-dígitos-octais, mas as outras seqüências de contrabarra listadas acima são utilizadas para estes caracteres de controle.

Nunca coloque uma contrabarra antes dos caracteres de dado `N` ou ponto (`.`), senão os dois serão interpretados, erroneamente, como a cadeia de caracteres nula padrão e o marcador de fim dos dados, respectivamente. Qualquer outro caractere não mencionado na tabela acima é interpretado como representando a si próprio.

É fortemente recomendado que os aplicativos que geram dados para o COPY convertam os caracteres de nova-linha e de retorno-de-carro presentes nos dados em seqüências `\n` e `\r` respectivamente. No presente momento (PostgreSQL 7.2 e versões mais antigas) é possível se representar um retorno-de-carro nos dados sem nenhuma seqüência especial, e representar a nova-linha nos dados por uma contrabarra seguida por um caractere de nova-linha. Entretanto, estas representações não serão aceitas por padrão nas versões futuras.

Observe que o fim de cada linha é marcado por um caractere de nova-linha no estilo Unix (“\n”). Atualmente, o `COPY FROM` não se comporta de forma adequada quando o arquivo especificado contém o fim de linha no estilo MS-DOS ou Mac. Espera-se que isto mude em versões futuras.

Formato Binário

O formato do arquivo usado pelo `COPY BINARY` mudou no PostgreSQL v7.1. O novo formato do arquivo consiste em uma cabeçalho, zero ou mais tuplas e um rodapé.

Cabeçalho do Arquivo

O cabeçalho do arquivo consiste de 24 bytes para campos fixos, seguidos por uma área de extensão do

cabeçalho de tamanho variável. Os campos fixos são:

Assinatura

A seqüência de 12 bytes `PGBCOPY\n\377\r\n\0` --- observe que o nulo é uma parte requerida da assinatura (A assinatura foi projetada para permitir a fácil identificação de arquivos corrompidos por uma transferência não apropriada. Esta assinatura é modificada por filtros de tradução de nova-linha, nulos suprimidos, bits altos suprimidos, ou mudanças de paridade).

Campo de disposição de inteiro

Constante int32 0x01020304 na ordem dos bytes de origem. Se uma ordem errada dos bytes for detectada aqui, o leitor poderá se envolver em uma mudança na ordem dos bytes dos campos posteriores.

Campo de sinalizadores

Máscara de bits int32 usada para caracterizar aspectos importantes do formato do arquivo. Os bits são numerados de 0 (LSB) até 31 (MSB) --- observe que este campo é armazenado na ordem dos bytes da plataforma de origem, assim como todos os campos inteiro seguintes. Os bits 16-31 são reservados para caracterizar questões críticas do formato do arquivo; o leitor deve abortar se for encontrado um bit não esperado neste intervalo. Os bits 0-15 são reservados para sinalizar questões de formato precedente-compatíveis; o leitor deve simplesmente ignorar qualquer bit não esperado neste intervalo. Atualmente somente um bit sinalizador está definido, os outros devem ser zero:

Bit 16

Se for igual a 1 os OIDs estão incluídos no arquivo; 0 caso contrário.

Comprimento da área de extensão do cabeçalho

Valor int32 do comprimento em bytes do restante do cabeçalho, não se incluindo. Na versão inicial será zero, com a primeira tupla vindo a seguir. Mudanças futuras no formato podem permitir a presença de dados adicionais no cabeçalho. O leitor deve simplesmente desprezar qualquer dado da área de extensão do cabeçalho que não souber o que fazer com a mesma.

A área de extensão do cabeçalho é imaginada como contendo uma seqüência de blocos auto-identificantes. O campo de sinalizadores não tem por finalidade informar os leitores o que existe na área de extensão. O projeto específico do conteúdo da área de extensão do cabeçalho foi deixado para uma versão futura.

Este projeto permite tanto adições de cabeçalho precedente-compatíveis (adicionar blocos de extensão de cabeçalho, ou sinalizar com bits de baixa-ordem) e mudanças não precedente-compatíveis (usar bits de alta-ordem para sinalizar estas mudanças, e adicionar dados de apoio para a área de extensão se for necessário).

Tuplas

Cada tupla começa com um contador (int16) do número de campos na tupla (atualmente todas as tuplas da tabela possuem o mesmo contador, mas isto pode não ser verdade para sempre). Então, repetido para

cada campo da tupla, existe uma palavra tipo-comprimento (`typlen`, int16) possivelmente seguida por um campo de dados. O campo tipo-comprimento é interpretado como:

Zero

O campo é nulo. Nenhum dado segue.

> 0

O campo possui um tipo de dado de comprimento fixo. Exatamente N bytes de dados seguem a palavra tipo-comprimento (`typlen`).

-1

O campo possui um tipo de dado de comprimento variável (`varlena`). Os próximos quatro bytes são o cabeçalho, que contém o valor do comprimento total incluindo a si próprio.

$<$ -1

Reservado para uso futuro.

Para campos não nulos, o leitor pode verificar se o tipo-comprimento (`typlen`) corresponde ao tipo-comprimento esperado para a coluna de destino, fornecendo uma maneira simples, mas muito útil, de verificar se o dado está como o esperado.

Não existe nenhum preenchimento de alinhamento ou qualquer dado extra entre os campos. Também observe que o formato não distingue se um tipo de dado é passado por referência ou passado por valor. Estas duas disposições são deliberadas: elas ajudam a melhorar a portabilidade dos arquivos (embora problemas relacionados com a ordem dos bytes ou o formato do ponto flutuante podem impedir a troca de arquivos binários entre computadores).

Se os OIDs são incluídos no arquivo, o campo OID segue imediatamente a palavra contador-de-campos. É um campo normal, exceto que não está incluído no contador-de-campos. Em particular possui um tipo-comprimento --- isto permite tratar OIDs de 4 bytes versus 8 bytes sem muita dificuldade, e permite os OIDs serem mostrados como NULL se por acaso for desejado.

Rodapé do Arquivo

O rodapé do arquivo consiste em uma palavra int16 contendo -1, sendo facilmente distinguível da palavra contador-de-campos da tupla.

O leitor deve relatar um erro se a palavra contador-de-campos não for -1 nem o número esperado de colunas, fornecendo uma verificação extra com relação a perda de sincronização com os dados.

Utilização

O exemplo a seguir copia uma tabela para a saída padrão, usando a barra vertical (|) como delimitador de campo:

```
COPY paises TO stdout USING DELIMITERS '|';
```

Para copiar dados de um arquivo Unix para a tabela `paises`:

```
COPY paises FROM '/usr1/proj/bray/sql/paises.txt';
```

Abaixo está um exemplo de dados apropriados para se copiar para a tabela a partir de `stdin` (por isso possui a seqüência de terminação na última linha):

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
ZM      ZAMBIA
ZW      ZIMBABWE
\.
```

Observe que os espaços em branco em cada linha são, na verdade, o caractere de tabulação.

Abaixo são os mesmos dados, escritos no formato binário em uma máquina Linux/i586. Os dados mostrados foram filtrados através do utilitário do Unix `od -c`. A tabela possui três campos: o primeiro é `char(2)`; o segundo é `text`; o terceiro é `integer`. Todas as linhas possuem um valor nulo no terceiro campo.

```
0000000   P   G   B   C   O   P   Y  \n 377  \r  \n  \0 004 003 002 001
0000020  \0  \0  \0  \0  \0  \0  \0  \0 003  \0 377 377 006  \0  \0  \0
0000040   A   F 377 377 017  \0  \0  \0   A   F   G   H   A   N   I   S
0000060   T   A   N  \0  \0 003  \0 377 377 006  \0  \0  \0   A   L 377
0000100 377  \v  \0  \0  \0   A   L   B   A   N   I   A  \0  \0 003  \0
0000120 377 377 006  \0  \0  \0   D   Z 377 377  \v  \0  \0  \0   A   L
0000140   G   E   R   I   A  \0  \0 003  \0 377 377 006  \0  \0  \0   Z
0000160   M 377 377  \n  \0  \0  \0   Z   A   M   B   I   A  \0  \0 003
0000200  \0 377 377 006  \0  \0  \0   Z   W 377 377  \f  \0  \0  \0   Z
0000220   I   M   B   A   B   W   E  \0  \0 377 377
```

Compatibilidade

SQL92

Não existe o comando `COPY` no SQL92.

CREATE AGGREGATE

Name

`CREATE AGGREGATE` — define uma nova função de agregação

Synopsis

```
CREATE AGGREGATE nome ( BASETYPE = tipo_dado_entrada,
    SFUNC = func_trans_estado, STYPE = tipo_dado_estado
    [ , FINALFUNC = func_final ]
    [ , INITCOND = cond_inicial ] )
```

Entradas

nome

O nome da função de agregação a ser criada.

tipo_dado_entrada

O tipo do dado de entrada sobre o qual esta função de agregação opera. Pode ser especificado como `"ANY"` para uma função de agregação que não examina seus valores de entrada (um exemplo é a função `count(*)`).

func_trans_estado

O nome da função de transição de estado a ser chamada para cada valor dos dados da entrada. Normalmente esta é uma função com dois argumentos, o primeiro sendo do tipo *tipo_dado_estado* e o segundo do tipo *tipo_dado_entrada*. Outra possibilidade, para uma função de agregação que não examina seus valores de entrada, é a função possuir apenas um argumento do tipo *tipo_dado_estado*. Em qualquer um dos casos a função deve retornar um valor do tipo *tipo_dado_estado*. Esta função recebe o valor atual do estado e o item atual de dado da entrada, e retorna o valor do próximo estado.

tipo_dado_estado

O tipo de dado do valor do estado da agregação.

func_final

O nome da função final chamada para calcular o resultado da agregação após todos os dados da entrada terem sido percorridos. A função deve possuir um único argumento do tipo *tipo_dado_estado*. O tipo de dado do valor da agregação é definido pelo tipo do valor retornado por esta função. Se *func_final* não for especificado, então o valor do estado final é utilizado como sendo o resultado da agregação, e o tipo da saída fica sendo o *tipo_dado_estado*.

cond_inicial

A atribuição inicial para o valor do estado. Deve ser uma constante literal na forma aceita pelo tipo de dado *tipo_dado_estado*. Se não for especificado, o valor do estado começa com NULL.

Saídas

`CREATE`

Mensagem retornada se o comando for executado com sucesso.

Descrição

O comando `CREATE AGGREGATE` permite ao usuário ou ao programador estender as funcionalidades do PostgreSQL definindo novas funções de agregação. Algumas funções de agregação para tipos base, como `min(integer)` e `avg(double precision)` estão presentes na distribuição base. Se forem definidos tipos novos, ou se houver a necessidade de uma função de agregação que não esteja presente, então o comando `CREATE AGGREGATE` pode ser utilizado para criar as funcionalidades desejadas.

Uma função de agregação é identificada pelo seu nome e pelo tipo de dado da entrada. Duas funções de agregação podem possuir o mesmo nome no caso de operarem sobre dados de entrada de tipos diferentes. Para evitar confusão, não crie uma função comum com o mesmo nome e tipo de dado de entrada de uma função de agregação.

Uma função de agregação é constituída de uma ou duas funções comuns: uma função de transição de estado *`func_trans_estado`*, e outra função, opcional, para a realização dos cálculos finais *`func_final`*. Estas funções são utilizadas da seguinte maneira:

```
func_trans_estado( estado-interno, próximo-item-dado ) ---> próximo-estado-interno
func_final( estado-interno ) ---> valor-da-agregação
```

O PostgreSQL cria uma variável temporária do tipo *`tipo_dado_estado`* para armazenar o estado interno atual da agregação. Para cada item de dado da entrada, a função de transição de estado é chamada para calcular o novo valor do estado interno. Após todos os dados terem sido processados, a função final é chamada uma vez para calcular o valor de saída da agregação. Se não houver nenhuma função final, então o valor do estado final é retornado.

A função de agregação pode possuir uma condição inicial, ou seja, um valor inicial para o valor de estado interno. Este valor é especificado e armazenado no banco de dados em um campo do tipo `text`, mas deve possuir uma representação externa válida de uma constante do tipo de dado do estado. Se não for especificado, então o valor do estado começa com NULL.

Se a função de transição de estado for declarada como "strict", então não poderá ser chamada com valores da entrada nulos. Para este tipo de função de transição, a execução da agregação é realizada da seguinte maneira. Valores da entrada nulos são ignorados (a função não é chamada e o valor do estado anterior permanece). Se o valor do estado inicial for nulo, então o primeiro valor de entrada que não for nulo

substitui o valor do estado, e a função de transição é chamada a partir do segundo valor de entrada que não for nulo. Este procedimento é útil para implementar funções de agregação como `max`. Observe que este comportamento somente está disponível quando o *`tipo_dado_estado`* for do mesmo tipo do *`tipo_dado_entrada`*. Quando estes tipos de dado forem diferentes, deverá ser fornecido um valor não nulo para a condição inicial, ou utilizar uma função de transição que não seja estrita.

Se a função de transição de estado não for estrita então será chamada, incondicionalmente, para cada valor da entrada, devendo ser capaz de lidar com valores nulos da entrada e valores nulos de transição. Esta opção permite ao autor da função de agregação ter pleno controle sobre os valores nulos.

Se a função final for declarada “strict”, estão não será chamada quando o valor do estado final for nulo; em vez disso, um resultado nulo será produzido automaticamente (É claro que este é apenas o comportamento normal de funções estritas). De qualquer forma, a função final tem sempre a opção de retornar nulo. Por exemplo, a função final para `avg` retorna nulo quando não há nenhum valor de entrada.

Notas

Use o comando `DROP AGGREGATE` para excluir funções de agregação.

Os parâmetros do comando `CREATE AGGREGATE` podem ser escritos em qualquer ordem, e não apenas na ordem mostrada acima.

Utilização

Consulte o capítulo sobre funções de agregação no *Guia do Programador do PostgreSQL* para ver exemplos completos sobre a sua utilização.

Compatibilidade

SQL92

O comando `CREATE AGGREGATE` é uma extensão do PostgreSQL à linguagem. Não existe o comando `CREATE AGGREGATE` no SQL92.

CREATE CONSTRAINT TRIGGER

Name

`CREATE CONSTRAINT TRIGGER` — define um novo gatilho de restrição

Synopsis

```
CREATE CONSTRAINT TRIGGER nome
    AFTER eventos ON
    relação restrição atributos
    FOR EACH ROW EXECUTE PROCEDURE função '(' args ')'
```

Entradas

nome

O nome do gatilho de restrição.

eventos

As categorias dos eventos para as quais este gatilho deve ser disparado.

relação

O nome da tabela que dispara o gatilho.

restrição

Especificação da restrição.

atributos

Atributos da restrição.

função(*args*)

Função a ser chamada como parte do processamento do gatilho.

Saídas

`CREATE CONSTRAINT`

Mensagem retornada se o comando for executado com sucesso.

Descrição

O comando `CREATE CONSTRAINT TRIGGER` é utilizado de dentro do comando `CREATE/ALTER TABLE` e pelo pg_dump para criar gatilhos especiais para a integridade referencial.

Não há a intenção de ser para uso geral.

CREATE DATABASE

Name

`CREATE DATABASE` — cria um banco de dados novo

Synopsis

```
CREATE DATABASE nome
    [ WITH [ LOCATION = 'caminho' ]
           [ TEMPLATE = gabarito ]
           [ ENCODING = codificação ] ]
```

Entradas

nome

O nome do banco de dados a ser criado.

caminho

Um local alternativo no sistema de arquivos onde será armazenado o banco de dados, especificado como uma cadeia de caracteres; ou `DEFAULT` para utilizar o local padrão.

gabarito

Nome do banco de dados a ser usado como gabarito para a criação do novo banco de dados, ou `DEFAULT` para utilizar o banco de dados de gabarito padrão (`template1`).

codificação

Método de codificação multibyte a ser utilizado no novo banco de dados. Especifique o nome como uma cadeia de caracteres (por exemplo, `'SQL_ASCII'`), ou o número inteiro da codificação, ou `DEFAULT` para utilizar a codificação padrão.

Saídas

`CREATE DATABASE`

Mensagem retornada se o comando for executado com sucesso.

`ERROR: user 'nome_do_usuário' is not allowed to create/drop databases`

É necessário possuir o privilégio especial `CREATEDB` para criar bancos de dados. Consulte o comando *CREATE USER*.

`ERROR: createdb: database "nome" already exists`

Esta mensagem ocorre se o banco de dados com o *nome* especificado já existir.

`ERROR: database path may not contain single quotes`

O local do banco de dados especificado em *`caminho`* não pode conter apóstrofos (`'`). Isto é necessário para que o interpretador de comandos, que vai criar o diretório do banco de dados, possa executar com segurança.

`ERROR: CREATE DATABASE: may not be called in a transaction block`

Havendo um bloco de transação explícito sendo executado não é possível utilizar o comando `CREATE DATABASE`. Primeiro a transação deve terminar.

`ERROR: Unable to create database directory '`*`caminho`*`'.`
`ERROR: Could not initialize database directory.`

Estas mensagens estão relacionadas com a falta de permissão no diretório de dados, o disco estar cheio, ou outro problema no sistema de arquivos. O usuário, sob o qual o servidor de banco de dados está processando, deve ter permissão para o local.

Descrição

O comando `CREATE DATABASE` cria um banco de dados novo no PostgreSQL. O criador se torna o dono do novo banco de dados.

Um local alternativo pode ser especificado com a finalidade de, por exemplo, armazenar o banco de dados em um disco diferente. O caminho deve ter sido preparado anteriormente com o comando *initlocation*.

Se o nome do caminho não possuir uma barra (/) é interpretado como sendo o nome de uma variável de ambiente, a qual deve ser conhecida pelo servidor. Desta maneira, o administrador do banco de dados pode exercer controle sobre os locais onde os bancos de dados podem ser criados (Uma escolha usual é, por exemplo, `PGDATA2`). Se o servidor for compilado com `ALLOW_ABSOLUTE_DBPATHS` (o que não é feito por padrão), nomes de caminhos absolutos, identificados por uma barra inicial (por exemplo, `/usr/local/pgsql/data`), também são permitidos.

Por padrão, o novo banco de dados será criado clonando o banco de dados padrão do sistema `template1`. Um gabarito diferente pode ser especificado escrevendo-se `TEMPLATE =` *`nome`*. Em particular, escrevendo-se `TEMPLATE = template0`, pode ser criado um banco de dados básico contendo apenas os objetos padrão predefinidos pela versão do PostgreSQL sendo utilizada. Esta forma é útil quando se deseja evitar a cópia de qualquer objeto da instalação local que possa ter sido adicionado ao `template1`.

O parâmetro opcional de codificação permite a escolha de uma codificação para o banco de dados, se o servidor em uso foi compilado com suporte à codificação multibyte. Quando este parâmetro não é especificado, o padrão é utilizar a mesma codificação do banco de dados usado como gabarito.

Os parâmetros opcionais podem ser escritos em qualquer ordem, e não apenas na ordem mostrada acima.

Notas

O comando `CREATE DATABASE` é uma extensão da linguagem do PostgreSQL.

Use o comando *DROP DATABASE* para excluir um banco de dados.

O aplicativo *createdb* é um script envoltório criado em torno deste comando, fornecido por conveniência.

Existem questões associadas à segurança e à integridade dos dados envolvidas na utilização de locais alternativos para os bancos de dados especificados por caminhos absolutos. Por isso, pelo padrão, somente uma variável de ambiente conhecida pelo gerenciador de banco de dados pode ser especificada para um local alternativo. Consulte o *Guia do Administrador do PostgreSQL* para obter mais informações.

Embora seja possível copiar outros bancos de dados além do `template1` especificando-se seu nome como gabarito, não se pretende (pelo menos ainda) que seja uma funcionalidade para COPY DATABASE de uso geral. É recomendado que os bancos de dados utilizados como gabarito sejam tratados como se fossem apenas de leitura. Consulte o *Guia do Administrador do PostgreSQL* para obter mais informações.

Utilização

Para criar um banco de dados novo:

```
silva=> create database lusiadas;
```

Para criar um banco de dados novo na área alternativa `~/bd_privado`:

```
$ mkdir bd_privado
$ initlocation ~/bd_privado
    The location will be initialized with username "silva".
This user will own all the files and must also own the server process.
Creating directory /home/silva/bd_privado
Creating directory /home/silva/bd_privado/base

initlocation is complete.
```

```
$ psql silva
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

silva=> CREATE DATABASE outrolugar WITH LOCATION = '/home/silva/bd_privado';
CREATE DATABASE
```

Compatibilidade

SQL92

Não existe o comando `CREATE DATABASE` no SQL92. Os bancos de dados são equivalentes aos catálogos cuja criação é definida pela implementação.

CREATE FUNCTION

Name

CREATE FUNCTION — define uma nova função

Synopsis

```
CREATE [ OR REPLACE ] FUNCTION nome ( [ tipo_do_argumento [, ...] ] )
    RETURNS tipo_retornado
    AS 'definição'
    LANGUAGE nome_ling
    [ WITH ( atributo [, ...] ) ]
CREATE [ OR REPLACE ] FUNCTION nome ( [ tipo_do_argumento [, ...] ] )
    RETURNS tipo_retornado
    AS 'arq_objeto', 'simbolo_de_ligação'
    LANGUAGE nome_ling
    [ WITH ( atributo [, ...] ) ]
```

Descrição

O comando `CREATE FUNCTION` define uma nova função. O comando `CREATE OR REPLACE FUNCTION` tanto cria uma nova função, quanto substitui uma função existente.

Parâmetros

nome

O nome da função a ser criada. Não é necessário que o nome seja único, porque as funções podem ser sobrecarregadas (overloaded), mas as funções que possuem o mesmo nome devem ter argumentos de tipos diferentes.

tipo_do_argumento

Os tipos de dado dos argumentos da função, caso existam. Os tipos de dado da entrada podem ser o tipo base, complexo, `opaque`, ou o mesmo tipo de uma coluna existente. `Opaque` indica que a função aceita argumentos de tipo não SQL, como o `char *`. O tipo de dado da coluna é referenciado escrevendo-se *nome_da_tabela.nome_da_coluna*`%TYPE`; utilizando-se esta notação ajuda a função se tornar independente das mudanças ocorridas na estrutura da tabela.

tipo_retornado

O tipo do dado retornado. O tipo de dado retornado pode ser especificado como um tipo base, complexo, `setof`, `opaque`, ou o mesmo tipo de uma coluna existente. O modificador `setof` indica que a função retorna um conjunto de itens, em vez de um único item. As funções em que se declara `opaque` para o valor do tipo retornado não retornam valor, e não podem ser chamadas diretamente; as funções dos gatilhos fazem uso desta funcionalidade.

definição

Uma cadeia de caracteres contendo a definição da função; o significado depende da linguagem. Pode ser o nome de uma função interna, o caminho para um arquivo objeto, uma consulta SQL, ou um texto escrito em uma linguagem procedural.

arq_objeto, *simbolo_de_ligação*

Esta forma da cláusula `AS` é usada para funções escritas na linguagem C, ligadas dinamicamente, quando o nome da função no código fonte da linguagem C não é o mesmo nome da função SQL. A cadeia de caracteres *arq_objeto* é o nome do arquivo contendo o objeto carregado dinamicamente, e *simbolo_de_ligação* é o símbolo de ligação do objeto, ou seja, o nome da função no código fonte escrito na linguagem C.

nome_ling

Pode ser `SQL`, `C`, `internal`, ou *nome_ling_proc*, onde *nome_ling_proc* é o nome de uma linguagem procedural criada. Consulte o comando *CREATE LANGUAGE* para obter mais detalhes. Para manter a compatibilidade com as versões anteriores, o nome pode estar entre apóstrofos (`'`).

atributo

Uma informação opcional sobre a função, utilizada para otimização. Veja abaixo para obter mais detalhes.

O usuário que cria a função torna-se o dono da função.

Os seguintes atributos podem estar presentes na cláusula WITH:

iscachable

O atributo `Iscachable` indica que a função sempre retorna o mesmo resultado quando recebe os mesmos valores para os argumentos (ou seja, não faz consultas ao banco de dados e também não utiliza informações que não estejam diretamente presentes na sua lista de parâmetros). O otimizador utiliza o atributo `iscachable` para saber se é seguro pré-executar a chamada da função.

isstrict

O atributo `isstrict` indica que a função sempre retorna NULL quando qualquer um de seus argumentos for nulo. Se este atributo for especificado, a função não será executada quando houver argumento nulo, ou seja, um resultado nulo será assumido automaticamente. Quando o atributo `isstrict` não for especificado, a função será chamada mesmo com argumentos de entrada nulos. Fica sendo responsabilidade do autor da função verificar valores nulos, se for necessário, e responder de forma apropriada.

Notas

Consulte o capítulo do *Guia do Programador do PostgreSQL* relativo à extensão do PostgreSQL através de funções para obter mais informações sobre como escrever funções externas.

A sintaxe completa do SQL é permitida para os argumentos de entrada e o valor retornado. Entretanto, alguns detalhes da especificação do tipo (por exemplo, a precisão para tipos `numeric`) são responsabilidade da implementação da função subjacente sendo silenciosamente aceitos pelo comando `CREATE FUNCTION` (ou seja, não são reconhecidos ou exigidos).

O PostgreSQL permite a *sobrecarga* de função, ou seja, o mesmo nome pode ser usado por várias funções diferentes desde que possuam argumentos com tipos diferentes. Entretanto, esta funcionalidade deve ser usada com cuidado para as funções internas e para as funções escritas na linguagem C.

Duas funções `internal` não podem possuir o mesmo nome no código C sem causar erros durante a fase de ligação. Para contornar este problema deve-se dar nomes diferentes no código C (por exemplo, usar os tipos dos argumentos como parte do nome no código C), então especificar este nome na cláusula AS do comando `CREATE FUNCTION`. Se a cláusula AS for omitida, então `CREATE FUNCTION` assume que o nome da função no código C tem o mesmo nome do SQL.

Analogamente, ao se sobrecarregar o nome das funções SQL através de várias funções escritas na linguagem C, deve ser dado a cada instância da função na linguagem C um nome distinto e, então, usar a forma alternativa da cláusula `AS` na sintaxe do `CREATE FUNCTION` para selecionar a implementação apropriada na linguagem C de cada função SQL sobrecarregada.

Quando chamadas repetidas ao comando `CREATE FUNCTION` fazem referência ao mesmo arquivo objeto, o arquivo só é carregado uma vez. Para descarregar e carregar o arquivo (talvez durante a fase de desenvolvimento), use o comando *LOAD*.

Use o comando `DROP FUNCTION` para remover as funções definidas pelo usuário.

Para atualizar a definição de uma função existente use o comando `CREATE OR REPLACE FUNCTION`. Observe que não é possível mudar o nome ou os tipos dos argumentos da função desta forma (se for tentado, será criada uma nova função distinta). O comando `CREATE OR REPLACE FUNCTION` também não permite que se mude o tipo do valor retornado de uma função existente. Para fazer isto, a função deve ser removida e recriada.

Se a função for removida e recriada, a nova função não é mais a mesma entidade que era antes; ficarão inválidas as regras, visões, gatilhos, etc... existentes que faziam referência à antiga função. Use o comando `CREATE OR REPLACE FUNCTION` para mudar a definição de uma função, sem invalidar os objetos que fazem referência à função.

Exemplos

Para criar uma função SQL simples:

```
CREATE FUNCTION um() RETURNS integer
    AS 'SELECT 1 AS RESULTADO;'
    LANGUAGE SQL;

SELECT um() AS resposta;
 resposta
----------
        1
```

Este exemplo cria uma função C chamando a rotina de uma biblioteca compartilhada criada pelo usuário chamada `funcs.so` (a extensão pode variar entre plataformas). O arquivo contendo a biblioteca compartilhada é procurado no caminho de procura de biblioteca compartilhada do servidor. Esta rotina em particular calcula o dígito verificador e retorna TRUE se o dígito verificador dos parâmetros da função está correto. A intenção é utilizar esta função numa restrição de verificação (CHECK).

```
CREATE FUNCTION ean_checkdigit(char, char) RETURNS boolean
    AS 'funcs' LANGUAGE C;

CREATE TABLE product (
    id        char(8) PRIMARY KEY,
    eanprefix char(8) CHECK (eanprefix ~ '[0-9]{2}-[0-9]{5}')
                      REFERENCES brandname(ean_prefix),
    eancode   char(6) CHECK (eancode ~ '[0-9]{6}'),
    CONSTRAINT ean    CHECK (ean_checkdigit(eanprefix, eancode))
);
```

No próximo exemplo é criada uma função que faz a conversão de tipo, do tipo complexo definido pelo usuário e o tipo nativo point. A função é implementada por um objeto carregado dinamicamente que foi compilado a partir de um fonte C (está ilustrada a alternativa obsoleta de se especificar o caminho absoluto para o arquivo contendo o objeto compartilhado). Para o PostgreSQL encontrar a função de conversão de tipo automaticamente, a função SQL deve ter o mesmo nome do tipo retornado, e por isso a sobrecarga é inevitável. O nome da função é sobrecarregado utilizando-se a segunda forma da cláusula `AS` na definição SQL:

```
CREATE FUNCTION point(complex) RETURNS point
    AS '/home/bernie/pgsql/lib/complex.so', 'complex_to_point'
    LANGUAGE C;
```

A declaração em C da função poderia ser:

```
Point * complex_to_point (Complex *z)
{
 Point *p;

 p = (Point *) palloc(sizeof(Point));
 p->x = z->x;
 p->y = z->y;

 return p;
}
```

Compatibilidade

O comando `CREATE FUNCTION` é definido no SQL99. A versão do PostgreSQL é similar mas não é totalmente compatível. Os atributos não são portáveis, nem as diferentes linguagens disponíveis o são.

Consulte também

DROP FUNCTION , LOAD, *Guia do Programador do PostgreSQL*

CREATE GROUP

Name

CREATE GROUP — define um novo grupo de usuários

Synopsis

```
CREATE GROUP nome [ [ WITH ] opção [ ... ] ]

onde opção pode ser:

     SYSID gid
   | USER  nome_usuário [, ...]
```

Entradas

nome

O nome do grupo.

gid

A cláusula SYSID pode ser utilizada para escolher o identificador do novo grupo no PostgreSQL. Entretanto, não há necessidade de ser utilizada.

Se esta cláusula não for especificada, o valor do identificador mais alto atribuído a um grupo, acrescido de um, começando por 1, será utilizado por padrão.

nome_usuário

A relação dos usuários a serem incluídos no grupo. Os usuários devem existir.

Saídas

CREATE GROUP

Mensagem retornada se o comando for executado com sucesso.

Descrição

O comando CREATE GROUP cria um novo grupo na instalação de banco de dados. Consulte o *Guia do Administrador do PostgreSQL* para obter mais informações sobre a utilização de grupos para autenticação. É necessário ser um superusuário do banco de dados para executar este comando.

Use o *ALTER GROUP* para incluir ou excluir usuários no grupo, e o *DROP GROUP* para excluir um grupo.

Utilização

Criar um grupo vazio:

```
CREATE GROUP engenharia;
```

Criar um grupo com membros:

```
CREATE GROUP vendas WITH USER jonas, marcela;
```

Compatibilidade

SQL92

Não existe o comando `CREATE GROUP` no SQL92. O conceito de "roles" é similar ao de grupos.

CREATE INDEX

Name

`CREATE INDEX` — define um índice novo

Synopsis

```
CREATE [ UNIQUE ] INDEX nome_do_índice ON tabela
    [ USING método_de_acesso ] ( coluna [ nome_do_operador ] [, ...] )
    [ WHERE predicado ]
CREATE [ UNIQUE ] INDEX nome_do_índice ON tabela
    [ USING método_de_acesso ] ( nome_da_função( coluna [, ... ]) [ nome_do_operador
    [ WHERE predicado ]
```

Entradas

UNIQUE

Faz com que o sistema procure por valores duplicados na tabela quando o índice é criado, se existirem dados na tabela, e sempre que novos dados forem adicionados. A tentativa de inserir ou de atualizar dados, que produza um valor duplicado, gera um erro.

nome_do_índice

O nome do índice a ser criado.

tabela

O nome da tabela a ser indexada.

método_de_acesso

O nome do método de acesso a ser utilizado pelo o índice. O método de acesso padrão é o BTREE. O PostgreSQL implementa quatro métodos de acesso para os índices:

BTREE

uma implementação das “B-trees” de alta concorrência de Lehman-Yao.

RTREE

implementa “R-trees” padrão, utilizando o algoritmo de partição quadrática de Guttman.

HASH

uma implementação das dispersões lineares de Litwin.

GIST

Generalized Index Search Trees (Árvores de Procura de Índice Generalizadas).

coluna

O nome de uma coluna da tabela.

nome_do_operador

Uma classe de operador associada. Veja abaixo para obter mais detalhes.

nome_da_função

Uma função que retorna um valor que pode ser indexado.

predicado

Define a expressão da restrição (constraint) para o índice parcial.

Saídas

`CREATE`

Mensagem retornada se o índice for criado com sucesso.

`ERROR: Cannot create index: 'nome_do_índice' already exists.`

Este erro ocorre se for impossível criar o índice.

Descrição

O comando `CREATE INDEX` constrói o índice *nome_do_índice* na *tabela* especificada.

> **Tip:** Os índices são utilizados, principalmente, para melhorar o desempenho do banco de dados, mas a utilização não apropriada causa uma degradação do desempenho.

Na primeira sintaxe exibida acima, os campos chave para o índice são especificados como nomes de coluna. Vários campos podem ser especificados, se o método de acesso do índice suportar índices com múltiplas colunas.

Na segunda sintaxe exibida acima, o índice é definido sobre o resultado da função definida pelo usuário *nome_da_função* aplicada sobre uma ou mais colunas de uma única tabela. Estes *índices funcionais* podem ser utilizados para obter acesso rápido aos dados baseado em operadores que normalmente iriam requerer alguma transformação para aplicá-los aos dados base.

O PostgreSQL implementa os métodos de acesso B-tree, R-tree, hash e GiST para os índices. O método de acesso B-tree é uma implementação das B-trees de alta concorrência de Lehman-Yao. O método de acesso R-tree implementa R-trees padrão utilizando o algoritmo de partição quadrática de Guttman. O

método de acesso hash é uma implementação das dispersões lineares de Litwin. Os algoritmos utilizados são mencionados apenas para informar que todos estes métodos de acesso são inteiramente dinâmicos, não necessitando de otimização periódica (como no caso de, por exemplo, métodos de acesso hash estáticos).

Quando a cláusula `WHERE` está presente, um *índice parcial* é criado. Um índice parcial é um índice que contém entradas apenas para uma parte da tabela, geralmente uma parte mais interessante do que o resto da tabela. Por exemplo, havendo uma tabela contendo tanto pedidos faturados quanto não faturados, onde os pedidos não faturados ocupam uma pequena fração da tabela, mas é a parte mais consultada, o desempenho pode ser melhorado criando-se um índice apenas para esta porção da tabela. Uma outra aplicação possível é a utilização da cláusula `WHERE` juntamente com `UNIQUE` para exigir a unicidade de um subconjunto dos dados da tabela.

A expressão utilizada na cláusula `WHERE` pode referenciar apenas as colunas da tabela subjacente (mas pode referenciar qualquer coluna, e não apenas as que estão sendo indexadas). Na forma atual, subconsultas e expressões de agregação não são permitidas na cláusula `WHERE`.

Todas as funções e operadores utilizados na definição de um índice devem ser *possíveis de serem armazenados na memória intermediária* (cachable), ou seja, seus resultados devem depender apenas de seus argumentos de entrada e nunca de uma influência externa (como o conteúdo de outra tabela ou a hora atual). Esta restrição garante que o comportamento do índice é bem definido. Para utilizar uma função definida pelo usuário em um índice deve ser utilizado o atributo 'Iscachable' na cláusula `WITH`.

Use o *DROP INDEX* para excluir um índice.

Notas

O otimizador de consultas do PostgreSQL vai considerar o uso de um índice B-tree sempre que um atributo indexado estiver envolvido em uma comparação utilizando um dos seguintes operadores: <, <=, =, >=, >

O otimizador de consultas do PostgreSQL vai considerar o uso de um índice R-tree sempre que um atributo indexado estiver envolvido em uma comparação utilizando um dos seguintes operadores: <<, &<, &>, >>, @, ~=, &&

O otimizador de consultas do PostgreSQL vai considerar o uso de um índice `hash` sempre que um atributo indexado estiver envolvido em uma comparação utilizando o operador `=`.

Atualmente somente os métodos de acesso `B-tree` e `Gist` suportam índices com mais de uma coluna. Por padrão, até 16 chaves podem ser especificadas (este limite pode ser alterado na geração do PostgreSQL). Na implementação atual, somente o `B-tree` suporta índices únicos.

Uma *classe de operador* pode ser especificada para cada coluna de um índice. A classe de operador identifica os operadores a serem utilizados pelo índice desta coluna. Por exemplo, um índice B-tree sobre inteiros de quatro bytes vai utilizar a classe de operadores `int4_ops`; esta classe de operadores inclui funções de comparação para inteiros de quatro bytes. Na prática, a classe de operadores padrão para o tipo de dado do campo é normalmente suficiente. O ponto principal em haver classes de operadores é que, para alguns tipos de dado, pode haver mais de uma ordenação que faça sentido. Por exemplo, pode se desejar ordenar o tipo de dado do número complexo tanto pelo valor absoluto, quanto pela parte real, o que pode ser feito definindo-se duas classes de operadores para o tipo de dado e, então, selecionando-se a classe apropriada para a construção do índice. Também existem algumas classes de operadores com finalidades especiais:

- As duas classes de operadores `box_ops` e `bigbox_ops` suportam índices R-tree para o tipo de dado `box`. A diferença entre as duas é que `bigbox_ops` ajusta as coordenadas da caixa para baixo, evitando exceções de ponto flutuante ao executar multiplicação, adição e subtração de coordenadas com números de ponto flutuante muito grande (Nota: isto era verdade há algum tempo atrás, mas atualmente as duas classes de operadores utilizam ponto flutuante e são efetivamente idênticas).

A seguinte consulta exibe todas as classes de operadores:

```
SELECT am.amname AS metodo_de_acesso,
       opc.opcname AS nome_do_operador,
       opr.oprname AS op_comparação
FROM   pg_am am, pg_opclass opc, pg_amop amop, pg_operator opr
WHERE  opc.opcamid = am.oid AND
       amop.amopclaid = opc.oid AND
       amop.amopopr = opr.oid
ORDER BY método_de_acesso, nome_do_operador, op_comparação;
```

Utilização

Para criar um índice `B-tree` para a coluna `titulo` na tabela `filmes`:

```
CREATE UNIQUE INDEX unq_titulo
    ON filmes (titulo);
```

Compatibilidade

SQL92

O comando CREATE INDEX é uma extensão do PostgreSQL à linguagem.

Não existe o comando `CREATE INDEX` no SQL92.

CREATE LANGUAGE

Name

`CREATE LANGUAGE` — define uma nova linguagem procedural

Synopsis

```
CREATE [ TRUSTED ] [ PROCEDURAL ] LANGUAGE nome_da_linguagem
    HANDLER tratador_de_chamadas
```

Descrição

Através do comando `CREATE LANGUAGE`, um usuário do PostgreSQL pode registrar uma nova linguagem procedural em um banco de dados do PostgreSQL. Depois, podem ser definidos funções e procedimentos de gatilhos nesta nova linguagem. O usuário deve possuir o privilégio de superusuário do PostgreSQL para poder registrar uma nova linguagem.

O comando `CREATE LANGUAGE` associa o nome da linguagem com o tratador de chamadas (call handler) que é responsável por executar as funções escritas nesta linguagem. Consulte o *Guia do Programador* para obter mais informações sobre os tratadores de chamadas das linguagens.

Observe que as linguagens procedurais são locais a cada bancos de dados. Para tornar uma linguagem disponível a todos os bancos de dados por padrão, esta linguagem deve ser instalada no banco de dados `template1`.

Parâmetros

`TRUSTED`

`TRUSTED` especifica que o tratador de chamadas para a linguagem é seguro, ou seja, não oferece a um usuário sem privilégios qualquer funcionalidade para contornar as restrições de acesso. Se esta palavra chave for omitida ao registrar a linguagem, somente usuários do PostgreSQL com privilégio de superusuário vão poder usar esta linguagem para criar novas funções.

`PROCEDURAL`

Apenas informativo.

`nome_da_linguagem`

O nome da nova linguagem procedural. Não existe distinção entre letras minúsculas e maiúsculas. Uma linguagem procedural não pode substituir uma das linguagens nativas do PostgreSQL.

Por compatibilidade com as versões anteriores, o nome pode ser escrito entre apóstrofos (`'`).

`HANDLER` *`tratador_de_chamadas`*

O *`tratador_de_chamadas`* é o nome de uma função, previamente registrada, que vai ser chamada para executar as funções escritas nesta linguagem procedural. O tratador de chamadas para a linguagem procedural deve ser escrito em uma linguagem compilada (como C), com a convenção

de chamadas versão 1, registrada no PostgreSQL como uma função que não recebe nenhum argumento e com retorno do tipo `opaque`, que é um substituto para tipos não especificados ou não definidos.

Diagnósticos

```
CREATE
```

Mensagem retornada se a linguagem for criada com sucesso.

```
ERROR:  PL handler function nome_da_função() doesn't exist
```

Este erro ocorre quando a função *nome_da_função*() não for encontrada.

Notas

Normalmente, este comando não deve ser executado diretamente pelos usuários. Para as linguagens procedurais fornecidas juntamente com a distribuição do PostgreSQL o aplicativo createlang deve ser utilizado, porque este aplicativo também instala o tratador de chamadas correto (O aplicativo `createlang` chama o `CREATE LANGUAGE` internamente).

Use o comando *CREATE FUNCTION* para criar uma nova função.

Use o comando *DROP LANGUAGE*, ou melhor ainda, o aplicativo droplang, para excluir linguagens procedurais.

O catálogo do sistema `pg_language` registra informações sobre as linguagens procedurais atualmente instaladas.

```
        Table "pg_language"
   Attribute   |  Type   | Modifier
---------------+---------+----------
 lanname       | name    |
 lanispl       | boolean |
 lanpltrusted  | boolean |
 lanplcallfoid | oid     |
 lancompiler   | text    |

   lanname   | lanispl | lanpltrusted | lanplcallfoid | lancompiler
-------------+---------+--------------+---------------+-------------
```

```
internal    | f       | f           |             0 | n/a
C           | f       | f           |             0 | /bin/cc
sql         | f       | f           |             0 | postgres
```

Atualmente, a definição de uma linguagem procedural não pode ser mudada após ter sido criada.

Exemplos

Os dois comandos mostrados abaixo, executados em seqüência, registram uma nova linguagem procedural e o tratador de chamadas associado:

```
CREATE FUNCTION minha_pl_trata_chamada () RETURNS opaque
    AS '$libdir/minha_pl'
    LANGUAGE C;
CREATE LANGUAGE minha_pl
    HANDLER minha_pl_trata_chamada;
```

Compatibilidade

O comando `CREATE LANGUAGE` é uma extensão do PostgreSQL à linguagem.

Histórico

O comando `CREATE LANGUAGE` apareceu pela primeira vez no PostgreSQL 6.3.

Consulte também

createlang, CREATE FUNCTION, droplang, DROP LANGUAGE, *Guia do Programador do PostgreSQL*

CREATE OPERATOR

Name

`CREATE OPERATOR` — define um novo operador

Synopsis

```
CREATE OPERATOR nome ( PROCEDURE = nome_da_função
     [, LEFTARG = tipo_esquerdo
     ] [, RIGHTARG = tipo_direito ]
     [, COMMUTATOR = op_comutador ] [, NEGATOR = op_negador ]
     [, RESTRICT = proc_restr ] [, JOIN = proc_juncao ]
     [, HASHES ] [, SORT1 = op_ord_esq ] [, SORT2 = op_ord_dir ] )
```

Entradas

nome

O operador a ser definido. Veja abaixo os caracteres permitidos.

nome_da_função

A função utilizada para implementar este operador.

tipo_esquerdo

O tipo do argumento do lado esquerdo do operador, se houver. Esta opção é omitida em um operador unário esquerdo.

tipo_direito

O tipo do argumento do lado direito do operador, se houver. Esta opção é omitida em um operador unário direito.

op_comutador

O comutador deste operador.

op_negador

O negador deste operador.

proc_restr

A função estimadora de seletividade de restrição para este operador.

proc_juncao

A função estimadora de seletividade de junção para este operador.

HASHES

Indica que este operador pode suportar uma junção por `hash`.

op_ord_esq

Se este operador pode suportar uma junção por mesclagem, o operador que ordena o tipo de dado do lado esquerdo deste operador.

op_ord_dir

Se este operador pode suportar uma junção por mesclagem, o operador que ordena o tipo de dado do lado direito deste operador.

Saídas

`CREATE`

Mensagem retornada se o operador for criado com sucesso.

Descrição

O comando `CREATE OPERATOR` define um novo operador *nome*. O usuário que define um operador se torna seu dono.

O operador *nome* é uma seqüência de até `NAMEDATALEN`-1 (31 por padrão) caracteres da seguinte lista:

+ - * / < > = ~ ! @ # % ^ & | ‘ ? $

Existem algumas restrições na escolha do nome:

- O `$` não pode ser definido como um operador de um único caractere, embora possa fazer parte do nome de um operador multi-caractere.
- As seqüências `--` e `/*` não podem aparecer em nenhum lugar do nome do operador, porque será considerado início de comentário.
- Um nome de operador multi-caractere não pode terminar por `+` ou por `-`, a menos que o nome também contenha pelo menos um dos seguintes caracteres:

  ~ ! @ # % ^ & | ‘ ? $

  Por exemplo, `@-` é um nome de operador permitido, mas `*-` não é. Esta restrição permite ao PostgreSQL analisar consultas em conformidade com o SQL sem requerer espaços entre elementos (tokens).

> **Note:** Ao se trabalhar com nomes de operadores que não sejam padrão SQL, usualmente é necessário usar espaços entre os operadores adjacentes para evitar ambigüidade. Por exemplo,

definindo-se um operador unário esquerdo chamado `@`, não se pode escrever `X*@Y`; deve-se escrever `X* @Y` para garantir que o PostgreSQL leia dois nomes de operadores e não um.

O operador `!=` é mapeado para <> na entrada, portanto estes dois nomes são sempre equivalentes.

Pelo menos um entre `LEFTARG` e `RIGHTARG` deve ser definido. Para operadores binários ambos devem ser definidos. Para operadores unários direito somente o LEFTARG deve ser definido, enquanto que para operadores unários esquerdo somente o RIGHTARG deve ser definido.

O procedimento *nome_da_função* deve ser previamente definido utilizando o comando `CREATE FUNCTION`, e deve estar definido para aceitar o número correto de argumentos (um ou dois) dos tipos indicados.

O operador comutador deve ser identificado, se existir, para que o PostgreSQL possa reverter a ordem dos operandos se desejar. Por exemplo, o operador area-menor-do-que, <<<, provavelmente teria o operador comutador area-maior-do-que, >>>. Com isso, o otimizador de consultas poderia livremente converter:

```
caixa '((0,0), (1,1))'  >>> minhas_caixas.descricao
```

em

```
minhas_caixas.descricao <<< caixa '((0,0), (1,1))'
```

Isto permite o código de execução sempre utilizar a última representação e simplifica um pouco o otimizador de consultas.

Analogamente, se existir um operador negador então este deve ser identificado. Suponha que exista um operador area-igual, ===, assim como um area-diferente, !==. A ligação negador permite ao otimizador de consultas simplificar

```
NOT minhas_caixas.descricao === caixa '((0,0), (1,1))'
```

para

```
minhas_caixas.descricao !== caixa '((0,0), (1,1))'
```

Se for fornecido o nome de um operador comutador, o PostgreSQL procura-o no catálogo. Se encontrá-lo, e este ainda não tiver um comutador próprio, então a entrada do comutador é atualizada para ter o novo operador criado como sendo o seu comutador. Aplica-se igualmente ao negador. Isto serve para permitir a definição de dois operadores que são o comutador ou o negador um do outro. O primeiro operador deve ser definido sem um comutador ou negador (conforme apropriado). Quando o segundo operador for definido, este nomeará o primeiro como seu comutador ou negador. O primeiro será atualizado como efeito colateral (A partir do PostgreSQL `6.5`, isto também funciona para se ter apenas dois operadores referindo um ao outro).

As opções HASHES, SORT1 e SORT2 estão presentes para apoiar o otimizador de consultas na realização de junções. O PostgreSQL sempre pode avaliar uma junção (i.e., processar uma cláusula com duas variáveis tuplas separadas por um operador que retorna um `booleano`) por substituição interativa [WONG76]. Adicionalmente, o PostgreSQL pode usar um algoritmo `junção-hash` junto com as linhas de [SHAP86]; entretanto, necessita saber se esta estratégia é aplicável. O algoritmo atual de `junção-hash` é correto apenas para operadores que representam testes de igualdade; além disso, a igualdade do tipo de dado deve significar igualdade bit a bit da representação do tipo (Por exemplo, um tipo de dado que contém bits não utilizados, que não fazem diferença nos testes de igualdade, não pode utilizar `junção-hash`). O sinalizador HASHES indica ao otimizador de consultas que a `junção-hash` pode ser usada com segurança com este operador.

Analogamente, os dois operadores de ordenação indicam ao otimizador de consultas se mesclagem-ordenação é uma estratégia de junção utilizável e quais operadores devem ser usados para ordenar as duas classes de operando. Os operadores de ordenação somente devem ser fornecidos para um operador de igualdade, devendo fazer referência aos operadores menor-do-que para os tipos de dado da esquerda e da direita, respectivamente.

Se forem descobertas outras estratégias de junção consideradas práticas, o PostgreSQL mudará o otimizador e o sistema de execução para usá-las, demandando especificação adicional quando um operador for definido. Felizmente, a comunidade de pesquisa não inventa novas estratégias de junção freqüentemente, e a generalidade adicionada por estratégias de junção definidas pelo usuário não foram consideradas como valendo a complexidade envolvida.

As opções `RESTRICT` e `JOIN` ajudam o otimizador de consultas a estimar o tamanho dos resultados. Se uma cláusula da forma:

```
minhas_caixas.descricao <<< caixa '((0,0), (1,1))'
```

estiver presente na qualificação, então o PostgreSQL poderá ter que estimar a fração das instâncias de `minhas_caixas` que satisfazem a cláusula. A função *`proc_restr`* deve ser uma função registrada (o que significa ter sido definida usando o comando `CREATE FUNCTION`) que aceita argumentos do tipo de dado correto e que retorna um número de ponto flutuante. O otimizador de consultas simplesmente chama esta função, passando o parâmetro `((0,0), (1,1))` e multiplica o resultado pelo tamanho da relação para obter o número esperado de instâncias.

Analogamente, quando os dois operandos do operador contêm variáveis instâncias, o otimizador deve estimar o tamanho da junção resultante. A função `proc_juncao` retornará outro número de ponto flutuante que será multiplicado pelas cardinalidades das duas tabelas envolvidas para calcular o tamanho esperado do resultado.

A diferença entre a função

```
meu_procedimento_1 (minhas_caixas.descricao, caixa '((0,0), (1,1))')
```

e o operador

```
minhas_caixas.descricao === caixa '((0,0), (1,1))'
```

é que o PostgreSQL tenta otimizar operadores e pode decidir usar um índice para restringir o espaço de procura quando operadores estão envolvidos. Entretanto, não existe tentativa de otimizar funções, que

são executadas pela força bruta. Mais ainda, as funções podem possuir qualquer número de argumentos, enquanto os operadores estão restritos a um ou dois.

Notas

Consulte o capítulo sobre operadores no *Guia do Usuário do PostgreSQL* para obter mais informações. Consulte o comando `DROP OPERATOR` para excluir do banco de dados um operador definido pelo usuário.

Utilização

O comando a seguir define o novo operador "area-igualdade" para o tipo de dado CAIXA:

```
CREATE OPERATOR === (
   LEFTARG = caixa,
   RIGHTARG = caixa,
   PROCEDURE = area_igual_procedimento,
   COMMUTATOR = ===,
   NEGATOR = !==,
   RESTRICT = area_restricao_procedimento,
   JOIN = area_juncao_procedimento,
   HASHES,
   SORT1 = <<<,
   SORT2 = >>>
);
```

Compatibilidade

SQL92

O comando `CREATE OPERATOR` é uma extensão do PostgreSQL à linguagem. Não existe o comando `CREATE OPERATOR` no SQL92.

CREATE RULE

Name

`CREATE RULE` — define uma nova regra

Synopsis

```
CREATE RULE nome AS ON evento
    TO objeto [ WHERE condição ]
    DO [ INSTEAD ] ação

onde ação pode ser:

NOTHING
|
consulta
|
( consulta ; consulta ... )
|
[ consulta ; consulta ... ]
```

Entradas

nome

O nome da regra a ser criada.

evento

Evento é um entre `SELECT`, `UPDATE`, `DELETE` e `INSERT`.

objeto

Objeto pode ser *tabela* ou *tabela*.*coluna* (Atualmente, somente a forma *tabela* está implementada).

condição

Qualquer expressão condicional-booleana SQL. A expressão de condição não pode fazer referência a nenhuma tabela, exceto `new` e `old`.

consulta

A consulta (ou consultas) causadora da *ação* pode ser um dos comandos SQL `SELECT`, `INSERT`, `UPDATE`, `DELETE`, ou `NOTIFY`.

Dentro da *condição* e da *ação* os nomes especiais de tabela `new` e `old` podem ser usados para se referir aos valores da tabela referenciada (o *objeto*). O `new` é válido nas regras ON INSERT e ON UPDATE

para se referir à nova linha sendo inserida ou atualizada. O `old` é válido nas regras ON UPDATE e ON DELETE para se referir à linha existente sendo atualizada ou excluída.

Saídas

`CREATE`

Mensagem retornada se a regra for criada com sucesso.

Descrição

O *sistema de regras* do PostgreSQL permite a definição de uma ação alternativa a ser realizada para as inclusões, atualizações ou exclusões nas tabelas do banco de dados. As regras também são utilizadas para implementar as visões das tabelas.

A semântica de uma regra é que, no instante em que uma instância individual (linha) é acessada, incluída, atualizada ou excluída, existe uma instância antiga (para consultas, atualizações e exclusões) e uma instância nova (para inclusões e atualizações). Toda as regras para o tipo de evento indicado no objeto de destino indicado (tabela) são examinadas, em uma ordem não especificada. Se a *`condição`* especificada na cláusula WHERE (caso exista) for verdadeira, a parte *`ação`* da regra é executada. A *`ação`* é executada em vez da consulta original se INSTEAD for especificado; senão é executada após a consulta original no caso de ON INSERT, ou antes da consulta original nos casos de ON UPDATE e ON DELETE. Dentro da *`condição`* e da *`ação`*, os valores dos campos da instância antiga e/ou da instância nova são substituídos por `old.`*`nome_atributo`* e `new.`*`nome_atributo`*.

A parte *`ação`* da regra pode ser composta por uma ou mais consultas. Para escrever várias consultas deve-se colocá-las entre parênteses ou entre colchetes. Estas consultas serão realizadas na ordem especificada (enquanto que não há garantia sobre a ordem de execução de múltiplas regras para o objeto). A *`ação`* também pode ser NOTHING indicando nenhuma ação. Portanto, a regra DO INSTEAD NOTHING suprime a execução da consulta original (quando sua condição for verdadeira); uma regra DO NOTHING não tem utilidade.

A parte *`ação`* da regra executa com o mesmo identificador do comando e da transação do comando do usuário que causou sua ativação.

Regras e Visões

Atualmente, as regras ON SELECT devem ser regras incondicionais INSTEAD e devem possuir ações compostas por uma única consulta SELECT. Portanto, uma regra ON SELECT torna efetivamente a tabela objeto em uma visão, cujo conteúdo visível são as linhas retornadas pela consulta SELECT da regra em vez de qualquer outra coisa que tenha sido armazenada na tabela (se houver). É considerado um estilo melhor escrever um comando CREATE VIEW do que criar uma tabela real e definir uma regra ON SELECT para esta tabela.

O comando CREATE VIEW cria uma tabela fictícia (sem armazenamento subjacente) e associa uma regra ON SELECT a esta tabela. O sistema não permite atualizações na visão, porque sabe que não existe

uma tabela real. Pode ser criada a ilusão de uma visão atualizável definindo-se regras para ON INSERT, ON UPDATE e ON DELETE (ou qualquer subconjunto destes comandos que for suficiente para atender as necessidades) para substituir as ações de atualização na visão por atualizações apropriadas em outras tabelas.

Existe um problema quando se tenta utilizar regras condicionais para a atualização das visões: é *obrigatório* haver uma regra incondicional INSTEAD para cada ação que se deseja permitir na visão. Se a regra for condicional, ou não for INSTEAD, então o sistema continuará rejeitando as tentativas de realizar uma ação de atualização, porque poderá tentar realizar a ação sobre a tabela fictícia em alguns casos. Se for desejado tratar todos os casos válidos através de regras condicionais, pode-se simplesmente adicionar uma regra incondicional DO INSTEAD NOTHING para garantir que o sistema sabe que nunca será chamado para atualizar a tabela fictícia. Em seguida devem ser criadas as regras condicionais como não INSTEAD; nos casos em que forem disparadas, vão se adicionar às ações padrão INSTEAD NOTHING.

Notas

É necessário possuir uma concessão para definição de regras na tabela para poder definir uma regra para a tabela. Use o comando `GRANT` e `REVOKE` para conceder e revogar as permissões.

É muito importante tomar cuidado para evitar as regras circulares. Por exemplo, embora qualquer uma destas duas definições de regra seja aceita pelo PostgreSQL, a consulta vai fazer com que o PostgreSQL retorne um erro porque a consulta vai circular muitas vezes:

```
CREATE RULE "_RETemp" AS
    ON SELECT TO emp
    DO INSTEAD
 SELECT * FROM toyemp;

CREATE RULE "_RETtoyemp" AS
    ON SELECT TO toyemp
    DO INSTEAD
 SELECT * FROM emp;
```

A tentativa de consultar a tabela `EMP` faz com que o PostgreSQL emita um erro porque a consulta vai circular muitas vezes:

```
SELECT * FROM emp;
```

Atualmente, se uma regra possui uma consulta NOTIFY, o NOTIFY será executado incondicionalmente --- ou seja, a notificação será emitida mesmo que não existam linhas onde a regra possa ser aplicada. Por exemplo, em

```
CREATE RULE me_notifique AS ON UPDATE TO minha_tabela DO NOTIFY minha_tabela;

UPDATE minha_tabela SET nome = ’foo’ WHERE id = 42;
```

um evento de notificação (NOTIFY) será emitido durante a atualização (UPDATE), existindo ou não alguma linha com id = 42. Esta é uma restrição da implementação que deverá estar corrigida em versões futuras.

Compatibilidade

SQL92

O comando `CREATE RULE` é uma extensão do PostgreSQL à linguagem. Não existe o comando `CREATE RULE` no SQL92.

CREATE SEQUENCE

Name

`CREATE SEQUENCE` — define um novo gerador de seqüência

Synopsis

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE nome_seq [ INCREMENT incremento ]
    [ MINVALUE valor_min ] [ MAXVALUE valor_max ]
    [ START início ] [ CACHE cache ] [ CYCLE ]
```

Entradas

TEMPORARY ou TEMP

Se for especificado, o objeto de seqüência é criado somente para esta sessão, e automaticamente eliminado ao término da sessão. Uma seqüência permanente com o mesmo nome, caso exista, não será visível nesta sessão enquanto a seqüência temporária existir.

nome_seq

O nome da seqüência a ser criada.

incremento

A cláusula `INCREMENT` *incremento* é opcional. Um valor positivo cria uma seqüência ascendente, enquanto um valor negativo cria uma seqüência descendente. O valor padrão é um (1).

valor_min

A cláusula opcional `MINVALUE` *valor_min* determina o valor mínimo que a seqüência pode gerar. Por padrão 1 e -2^63-1 para seqüências ascendentes e descendentes, respectivamente.

valor_max

A cláusula opcional `MAXVALUE` *valor_max* determina o valor máximo para a seqüência. Por padrão 2^63-1 e -1 para seqüências ascendentes e descendentes, respectivamente.

início

A cláusula opcional `START` *início* permite a seqüência iniciar com qualquer valor. Por padrão, o valor inicial é igual a *valor_min* para seqüências ascendentes, e igual a *valor_max* para seqüências descendentes.

cache

A opção `CACHE` *cache* permite que os números da seqüência sejam préviamente alocados e armazenados em memória para acesso mais rápido. O valor mínimo é 1 (somente um valor pode ser gerado de cada vez, ou seja, sem armazenamento em memória) e este também é o valor padrão.

CYCLE

A palavra chave opcional CYCLE pode ser utilizada para permitir a seqüência reiniciar quando o `valor_max` ou o `valor_min` for atingido pela seqüência ascendente ou descendente, respectivamente. Se o limite for atingido, o próximo número gerado será `valor_min` ou `valor_max`, respectivamente. Sem a cláusula CYCLE, após o limite ser atingido as chamadas à função `nextval` retornam um erro.

Saídas

`CREATE`

Mensagem retornada se o comando for executado com sucesso.

`ERROR: Relation 'nome_seq' already exists`

A seqüência especificada já existe.

`ERROR: DefineSequence: MINVALUE (início) can't be >= MAXVALUE (max)`

O valor especificado para o início da seqüência está fora do intervalo.

`ERROR: DefineSequence: START value (início) can't be < MINVALUE (min)`

O valor especificado para o início da seqüência está fora do intervalo.

`ERROR: DefineSequence: MINVALUE (min) can't be >= MAXVALUE (max)`

Os valores mínimo e máximo estão inconsistentes.

Descrição

O comando `CREATE SEQUENCE` cria um novo gerador de números seqüenciais no banco de dados em uso. Este comando envolve a criação e a inicialização de uma tabela nova com uma única linha com o nome `nome_seq`. O usuário que executa o comando se torna o dono do gerador.

Após a seqüência ser criada, podem ser utilizadas as funções `nextval`, `currval` e `setval` para trabalhar com a seqüência. Estas funções estão documentadas no *Guia do Usuário*.

Embora não seja possível atualizar uma seqüência diretamente, é possível realizar uma consulta do tipo

```
SELECT * FROM nome_seq;
```

para conhecer os parâmetros e o estado atual da seqüência. Em particular, o campo `last_value` da seqüência mostra o último valor alocado por qualquer processo servidor (É claro que este valor pode estar obsoleto na hora em que for exibido, se outros processos estiverem chamando a função `nextval`).

> **Caution**
>
> Podem ocorrer resultados não esperados ao se especificar um valor maior do que 1 para *cache* em um objeto de seqüência utilizado ao mesmo tempo por vários servidores (backends). Cada servidor aloca e armazena em memória valores sucessivos da seqüência ao fazer um único acesso ao objeto de seqüência e incrementa o último valor (`last_value`) na forma correspondente. Então, a próxima utilização de *cache*-1 da função `nextval` neste servidor, simplesmente retorna os valores pré-alocados, sem tocar no objeto compartilhado. Desta forma, todos os valores alocados nesta sessão, mas não utilizados, são perdidos ao final da sessão. Além disso, embora seja garantido que os diversos servidores alocam valores distintos da seqüência, os valores podem ser gerados fora de seqüência quando são levados em conta todos os servidores. (Por exemplo, com um *cache* de 10, o servidor A pode reservar os valores de 1 a 10 e usar o próximo valor igual a 1, então o servidor B pode reservar os valores de 11 a 20 e usar o próximo valor igual a 11 antes do servidor A ter usado o próximo valor igual a 2). Assim, com um valor para *cache* igual a 1 é seguro assumir que os valores da função `nextval` são gerados seqüencialmente; com um valor para *cache* maior do que 1 pode-se assumir que os valores da função `nextval` são todos distintos, mas não que sejam gerados de forma puramente seqüencial. Além disso, o valor do campo `last_value` reflete o último valor reservado por qualquer servidor, independentemente de ter sido ou não retornado por uma chamada à função `nextval`. Outra consideração a ser feita, é que a função `setval` executada neste tipo de seqüência não vai fazer efeito em outro servidor até que este tenha utilizado todos os valores pré-alocados em memória.

Notas

Use o comando `DROP SEQUENCE` para excluir uma seqüência.

As seqüências são baseadas em aritmética de tipo `bigint`, por isso a faixa de valores não pode ultrapassar a faixa permitida para números inteiros de 8 bytes (-9223372036854775808 a 9223372036854775807). Em algumas plataformas mais antigas pode não haver suporte do compilador para números inteiros de 8 bytes e, neste caso, as seqüências utilizam aritmética para o tipo regular `integer` (faixa de valores de -2147483648 a +2147483647).

Quando o valor para *cache* é maior do que 1, cada servidor utiliza sua própria memória para armazenar os números préviamente alocados. Os números armazenados em memória, mas não utilizados pela sessão atual, são perdidos, ocasionando uma seqüência cheia de “buracos”.

Utilização

Criar uma seqüência ascendente chamada `serial`, começando por 101:

```
CREATE SEQUENCE serial START 101;
```

Selecionar o próximo valor desta seqüência:

```
SELECT nextval('serial');
```

```
nextval
-------
    114
```

Utilizar esta seqüência em um comando INSERT:

```
INSERT INTO distribuidores VALUES (nextval('serial'), 'nada');
```

Atualizar o valor da seqüência após executar o comando COPY FROM:

```
BEGIN;
    COPY distribuidores FROM 'arquivo_entrada';
    SELECT setval('serial', max(id)) FROM distribuidores;
END;
```

Compatibilidade

SQL92

O comando `CREATE SEQUENCE` é uma extensão do PostgreSQL à linguagem. Não existe o comando `CREATE SEQUENCE` no SQL92.

CREATE TABLE

Name

CREATE TABLE — define uma nova tabela

Synopsis

```
CREATE [ [ LOCAL ] { TEMPORARY | TEMP } ] TABLE nome_da_tabela (
    { nome_da_coluna tipo_de_dado [ DEFAULT expressão_padrão ] [ restrição_de_coluna
    | restrição_de_tabela }  [, ... ]
)
[ INHERITS ( tabela_ascendente [, ... ] ) ]
[ WITH OIDS | WITHOUT OIDS ]

onde restrição_de_coluna é:

[ CONSTRAINT nome_da_restrição ]
{ NOT NULL | NULL | UNIQUE | PRIMARY KEY |
  CHECK (expressão) |
  REFERENCES tabela_referenciada [ ( coluna_referenciada ) ] [ MATCH FULL | MATCH PAR
    [ ON DELETE ação ] [ ON UPDATE ação ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

e restrição_de_tabela é:

[ CONSTRAINT nome_da_restrição ]
{ UNIQUE ( nome_da_coluna [, ... ] ) |
  PRIMARY KEY ( nome_da_coluna [, ... ] ) |
  CHECK ( expressão ) |
  FOREIGN KEY ( nome_da_coluna [, ... ] ) REFERENCES tabela_referenciada [ ( coluna_r
    [ MATCH FULL | MATCH PARTIAL ] [ ON DELETE ação ] [ ON UPDATE ação ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

Descrição

O comando CREATE TABLE cria uma tabela nova, inicialmente vazia, no banco de dados atual. O usuário que executa o comando torna-se o dono da tabela.

O comando CREATE TABLE também cria, automaticamente, um tipo de dado que representa o tipo da tupla (tipo estrutura) correspondente a uma linha da tabela. Portanto, uma tabela não pode ter o mesmo nome de um tipo de dado existente.

Uma tabela não pode possuir mais de 1600 colunas (Na prática, o limite efetivo é menor, devido à restrição do comprimento das tuplas). Uma tabela não pode ter o mesmo nome de uma tabela do catálogo do sistema.

As cláusulas opcionais de restrição especificam as restrições (ou testes) que as linhas novas ou modificadas devem satisfazer para a operação de inclusão ou de alteração ser completada. Uma restrição é uma regra

com nome: um objeto SQL que ajuda a definir conjuntos válidos de valores, estabelecendo limites nos resultados das operações de inclusão, exclusão e atualização realizadas na tabela.

Existem duas formas para definir restrições: restrições de tabela e restrições de coluna. A restrição de coluna é definida como parte da definição da coluna. A restrição de tabela não está presa a uma coluna em particular, podendo abranger mais de uma coluna. Toda restrição de coluna também pode ser escrita como uma restrição de tabela; a restrição de coluna é somente uma notação conveniente, no caso da restrição afetar apenas uma coluna.

Parâmetros

`[LOCAL] TEMPORARY` ou `[LOCAL] TEMP`

Se for especificado, a tabela é criada como uma tabela temporária. Tabelas temporárias são automaticamente eliminadas no final da sessão. Uma tabela permanente com o mesmo nome, caso exista, não será visível na sessão corrente enquanto a tabela temporária existir. Todo índice criado em tabela temporária também é temporário.

A palavra `LOCAL` é opcional. Veja sob *Compatibilidade*.

`nome_da_tabela`

O nome da tabela a ser criada.

`nome_da_coluna`

O nome da coluna a ser criada na nova tabela.

`tipo_de_dado`

O tipo de dado da coluna. Pode incluir especificadores de `array`. Consulte o *Guia do Usuário* para obter mais informações sobre tipos de dado e sobre `arrays`.

`DEFAULT` *`expressão_padrão`*

A cláusula `DEFAULT` atribui um valor padrão para o dado da coluna em cuja definição está presente. O valor pode ser qualquer expressão (subconsultas e referências cruzadas para outras colunas da mesma tabela não são permitidas). O tipo de dado da expressão padrão deve corresponder ao tipo de dado da coluna.

A expressão é utilizada em todas as operações de inclusão que não especificam valor para a coluna. Não havendo valor padrão para a coluna, então NULL torna-se o valor padrão.

`INHERITS (` *`tabela_ascendente`* `[, ... ] )`

A cláusula opcional `INHERITS` (herda) especifica uma lista de tabelas das quais a nova tabela herda, automaticamente, todas as colunas. Se o mesmo nome de coluna existir em mais de uma tabela ascendente um erro é gerado, a menos que o tipo de dado das colunas seja o mesmo em todas as tabelas ascendentes. Se não houver conflito, então as colunas duplicadas são mescladas para formar uma única coluna da nova tabela. Se a lista de nomes de colunas da nova tabela contém uma coluna que também é herdada, da mesma forma o tipo de dado deve ser o mesmo das colunas herdadas, e a definição das colunas será mesclada em uma única coluna. Entretanto, declarações de colunas novas e herdadas com o mesmo nome não precisam especificar restrições idênticas: todas as restrições fornecidas em qualquer uma das declarações são mescladas, sendo todas aplicadas à nova tabela. Se a nova tabela especificar, explicitamente, um valor padrão para a coluna, este valor padrão substitui

qualquer valor padrão das declarações herdadas. Fora isso, toda tabela ascendente que especificar um valor padrão para a coluna deve especificar o mesmo valor, ou um erro será gerado.

`WITH OIDS` ou `WITHOUT OIDS`

Esta cláusula opcional especifica se as linhas da nova tabela devem possuir OIDs (identificadores do objeto). O padrão é possuir OIDs (Se a nova tabela herdar de qualquer tabela que possua OIDs, então a cláusula `WITH OIDS` é forçada mesmo que no comando esteja especificado `WITHOUT OIDS`).

A especificação de `WITHOUT OIDS` permite ao usuário suprimir a geração dos OIDs para as linhas da tabela. Este procedimento pode ser interessante em tabelas grandes, porque reduz o consumo de OIDs e, portanto, adia o recomeço deste contador de 32 bits. Quando o contador recomeça, não é mais possível assumir a sua unicidade, o que reduz muito a sua utilidade.

`CONSTRAINT` *`nome_da_restrição`*

Um nome opcional para a restrição da coluna ou da tabela. Se não for especificado, o nome será gerado pelo sistema.

`NOT NULL`

Valores nulos não são permitidos na coluna. Esta declaração é equivalente à restrição de coluna `CHECK (`*`nome_da_coluna`*` NOT NULL)`.

`NULL`

Valores nulos são permitidos na coluna. Este é o padrão.

Esta cláusula só está disponível por compatibilidade com bancos de dados SQL fora do padrão. Sua utiização é desestimulada em novas aplicações.

`UNIQUE` (restrição da coluna)
`UNIQUE (` *`nome_da_coluna`* `[, ... ] )` (restrição da tabela)

A restrição `UNIQUE` especifica a regra onde um grupo de uma ou mais colunas distintas de uma tabela podem conter apenas valores únicos. O comportamento da restrição de unicidade para tabelas é o mesmo da restrição de unicidade para colunas, porém com a capacidade adicional de abranger várias colunas.

Para a finalidade da restrição de unicidade, valores nulos não são considerados iguais.

Cada restrição de unicidade da tabela deve abranger um conjunto de colunas diferente do conjunto de colunas abrangido por qualquer outra restrição de unicidade e da chave primária definida para a tabela (Senão, seria apenas a mesma restrição declarada duas vezes).

`PRIMARY KEY` (restrição da coluna)
`PRIMARY KEY (` *`nome_da_coluna`* `[, ... ] )` (restrição da tabela)

A restrição de chave primária especifica que a coluna, ou colunas, da tabela pode conter apenas valores únicos (não duplicados) e não nulos. Tecnicamente a chave primária (`PRIMARY KEY`) é simplesmente uma combinação de unicidade (`UNIQUE`) com não nulo (`NOT NULL`), mas identificar um conjunto de colunas como chave primária também fornece metadados sobre o projeto do esquema, porque chaves primárias indicam que outras tabelas podem depender deste conjunto de colunas como um identificador único para linhas.

Somente uma chave primária pode ser especificada para uma tabela, seja como uma restrição de coluna ou como uma restrição de tabela.

A restrição de chave primária deve abranger um conjunto de colunas que seja diferente de outro conjunto de colunas abrangido por uma restrição de unicidade definida para a mesma tabela.

`CHECK (`*`expressão`*`)`

A cláusula `CHECK` especifica restrições de integridade, ou testes, que as linhas novas ou atualizadas devem atender para que uma operação de inserção ou de atualização complete. Cada restrição deve ser uma expressão que produza um resultado booleano. Uma condição declarada na definição da coluna deve fazer referência apenas ao valor desta coluna, enquanto uma condição declarada como uma restrição da tabela pode fazer referência a várias colunas.

Atualmente, as expresões de `CHECK` não podem conter subconsultas, nem fazer referência a variáveis que não sejam colunas da linha atual.

`REFERENCES` *`tabela_referenciada`* `[ (` *`coluna_referenciada`* `) ] [ MATCH` *`tipo_corresp`* `] [ ON DELETE` *`ação`* `] [ ON UPDATE` *`ação`* `]` (restrição da coluna)
`FOREIGN KEY (` *`nome_da_coluna`* `[, ... ] ) REFERENCES` *`tabela_referenciada`* `[ (` *`coluna_referenciada`* `[, ... ] ) ] [ MATCH` *`tipo_corresp`* `] [ ON DELETE` *`ação`* `] [ ON UPDATE` *`ação`* `]` (restrição da tabela)

A restrição `REFERENCES` especifica que um grupo de uma ou mais colunas da nova tabela deve conter somente valores correspondentes aos valores das colunas referenciadas *`coluna_referenciada`* da tabela referenciada *`tabela_referenciada`*. Se a *`coluna_referenciada`* for omitida, a chave primária da *`tabela_referenciada`* é utilizada. As colunas referenciadas devem pertencer a uma restrição de chave primária ou de unicidade da tabela referenciada.

Os valores adicionados a estas colunas são comparados com os valores das colunas referenciadas da tabela referenciada utilizando o tipo de comparação. Existem 3 tipos de comparação: `MATCH FULL`, `MATCH PARTIAL` e o tipo de comparação padrão se nada for especificado. `MATCH FULL` não permite que uma coluna de uma chave estrangeira com várias colunas seja nula, a menos que todas as colunas da chave estrangeira sejam nulas. O tipo de comparação padrão permite que algumas colunas da chave estrangeira sejam nulas enquanto outras colunas da chave estrangeira não são nulas. `MATCH PARTIAL` ainda não está implementado.

Adicionalmente, quando os dados das colunas referenciadas são modificados, certas ações são realizadas nos dados das colunas desta tabela. A cláusula `ON DELETE` especifica a ação a ser executada quando uma linha referenciada da tabela referenciada é excluída. Da mesma maneira, a cláusula `ON UPDATE` especifica a ação a ser executada quando uma coluna referenciada da tabela referenciada muda de valor. Se a linha é atualizada, mas a coluna referenciada não muda de valor, nenhuma ação é executada. São possíveis as seguintes ações para cada cláusula:

`NO ACTION`

Gera um erro indicando que a exclusão ou a atualização criaria uma violação de chave estrangeira. Esta é a ação padrão.

`RESTRICT`

O mesmo que `NO ACTION`.

`CASCADE`

Exclui qualquer linha que faça referência à linha excluída, ou atualiza o valor da coluna que faz referência usando o novo valor da coluna referenciada, respectivamente.

`SET NULL`

Atribui nulo aos valores das colunas que fazem referência.

`SET DEFAULT`

Atribui o valor padrão às colunas que fazem referência.

Se a coluna da chave primária é atualizada freqüentemente, pode ser útil adicionar um índice às colunas da cláusula `REFERENCES`, para que as ações `NO ACTION` e `CASCADE` associadas às colunas da cláusula `REFERENCES` sejam executadas mais rapidamente.

`DEFERRABLE` ou `NOT DEFERRABLE`

Estas cláusulas controlam se as restrições podem ser postergadas. Uma restrição que não pode ser postergada é verificada imediatamente após cada comando. A verificação das restrições que são postergáveis pode ser adiada para o final da transação (usando o comando *SET CONSTRAINTS*). O padrão é `NOT DEFERRABLE`. Somente restrições de chave estrangeira aceitam esta cláusula no momento. Todos os outros tipos de restrição não são postergáveis.

`INITIALLY IMMEDIATE` ou `INITIALLY DEFERRED`

Se uma restrição é postergável, esta cláusula especifica o momento padrão para verificar a restrição. Se a restrição é `INITIALLY IMMEDIATE`, então é verificada após cada declaração. Este é o padrão. Se a declaração é `INITIALLY DEFERRED`, então é verificada apenas no final da transação. O momento de verificação da restrição pode ser alterado pelo comando *SET CONSTRAINTS*.

Diagnósticos

CREATE

Mensagem retornada se a tabela for criada com sucesso.

ERROR

Mensagem retornada se a criação da tabela falhar. Esta mensagem é normalmente acompanhada por algum texto descritivo, como: `ERROR: Relation '`*`nome_da_tabela`*`' already exists`, que ocorre quando a tabela especificada existe no banco de dados.

Notas

- Sempre que uma aplicação faz uso dos OIDs para identificar linhas específicas de uma tabela, é recomendado criar uma restrição de unicidade para a coluna `oid` da tabela, para garantir que os OIDs na tabela realmente identificam unicamente uma linha, mesmo após o contador recomeçar. Evite assumir que os OIDs são únicos entre tabelas; se for necessário um identificador único para todo o banco de dados, use uma combinação do `tableoid` (OID da tabela) com o OID da linha para esta finalidade (é provável que nas versões futuras do PostgreSQL existam contadores OID separados para cada tabela, então será *necessário*, e não opcional, incluir o `tableoid` para ter-se um identificador único para todo o banco de dados).

  Tip: O uso de `WITHOUT OIDS` não é recomendado para tabelas sem chave primária porque sem um OID, e sem uma chave de dados única, fica difícil identificar uma linha específica.

- O PostgreSQL cria, automaticamente, um índice para cada restrição de unicidade e de chave primária para garantir a sua unicidade. Portanto, não é necessário criar um índice explícito para as colunas da chave primária. (Consulte o comando *CREATE INDEX* para obter mais informações)
- O padrão SQL92 especifica que as restrições `CHECK` de colunas podem referenciar apenas a coluna à qual se aplica; somente restrições `CHECK` de tabelas podem fazer referências a várias colunas. O PostgreSQL não impõe esta restrição; as restrições de tabela e de colunas são tratadas da mesma maneira.
- As restrições de unicidade e de chave primária não são herdadas na implementação atual, tornando o comportamento da combinação de herança com restrição de unicidade diferente do esperado.

Exemplos

Criar a tabela `filmes` e a tabela `distribuidores`:

```
CREATE TABLE filmes (
    cod         CHARACTER(5) CONSTRAINT pk_filmes PRIMARY KEY,
    titulo      CHARACTER VARYING(40) NOT NULL,
    did         DECIMAL(3) NOT NULL,
    data_prod   DATE,
    tipo        CHAR(10),
    duracao     INTERVAL HOUR TO MINUTE
);

CREATE TABLE distribuidores (
     did    DECIMAL(3) PRIMARY KEY DEFAULT NEXTVAL('serial'),
     nome   VARCHAR(40) NOT NULL CHECK (nome <> '')
);
```

Criar uma tabela com uma matriz de 2 dimensões

```
CREATE TABLE matriz2d (
```

```
    matriz  INT[][]
);
```

Definir uma restrição de unicidade para a tabela filmes. Restrições de unicidade de tabela podem ser definidas usando uma ou mais colunas da tabela:

```
CREATE TABLE filmes (
    cod         CHAR(5),
    titulo      VARCHAR(40),
    did         DECIMAL(3),
    data_prod   DATE,
    tipo        VARCHAR(10),
    duracao     INTERVAL HOUR TO MINUTE,
    CONSTRAINT producao UNIQUE(data_prod)
);
```

Definir uma restrição de coluna para verificação:

```
CREATE TABLE distribuidores (
    did     DECIMAL(3) CHECK (did > 100),
    nome    VARCHAR(40)
);
```

Definir uma restrição de tabela para verificação:

```
CREATE TABLE distribuidores (
    did     DECIMAL(3),
    nome    VARCHAR(40)
    CONSTRAINT chk_dist CHECK (did > 100 AND nome <> ”)
);
```

Definir uma restrição de chave primária para a tabela `filmes`. Restrições de chave primária da tabela podem ser definidas usando uma ou mais colunas da tabela.

```
CREATE TABLE filmes (
    cod         CHAR(5),
    titulo      VARCHAR(40),
    did         DECIMAL(3),
    data_prod   DATE,
    tipo        VARCHAR(10),
    duracao     INTERVAL HOUR TO MINUTE,
    CONSTRAINT pk_filmes PRIMARY KEY(cod,titulo)
);
```

Definir a restrição de chave primária para a tabela `distribuidores`. Os dois exemplos abaixo são equivalentes, o primeiro utiliza a sintaxe de restrição de tabela, e o segundo utiliza a notação de restrição de coluna.

```
CREATE TABLE distribuidores (
    did     DECIMAL(3),
    nome    CHAR VARYING(40),
    PRIMARY KEY(did)
);

CREATE TABLE distribuidores (
    did     DECIMAL(3) PRIMARY KEY,
    nome    VARCHAR(40)
);
```

O comando abaixo especifica uma constante literal como valor padrão da coluna `nome`; faz com que o valor padrão da coluna `did` seja gerado como o próximo valor de um objeto de seqüência; faz com que o valor padrão da coluna `data_ins` seja a data e hora em que a linha é inserida.

```
CREATE TABLE distribuidores (
    nome      VARCHAR(40) DEFAULT 'luso filmes',
    did       INTEGER DEFAULT NEXTVAL('seq_distribuidores'),
    data_ins  TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Definir duas restrições de coluna `NOT NULL` na tabela `distribuidores`, sendo que uma delas recebe um nome explicito:

```
CREATE TABLE distribuidores (
    did     DECIMAL(3) CONSTRAINT nao_nulo NOT NULL,
    nome    VARCHAR(40) NOT NULL
);
```

Definir uma restrição de unicidade para a coluna `nome`:

```
CREATE TABLE distribuidores (
    did     DECIMAL(3),
    nome    VARCHAR(40) UNIQUE
);
```

O comando acima é equivalente ao mostrado abaixo, especificado através de uma restriçao de tabela:

```
CREATE TABLE distribuidores (
    did     DECIMAL(3),
    nome    VARCHAR(40),
    UNIQUE(nome)
);
```

Compatibilidade

O comando `CREATE TABLE` está em conformidade com o “SQL92 Intermediate” e com um subconjunto do SQL99, com excessões listadas abaixo e nas descrições acima.

Tabelas temporárias

Além da tabela temporária local, o SQL92 também define a declaração `CREATE GLOBAL TEMPORARY TABLE`. Tabelas temporárias globais também são visíveis por outras sessões.

Para as tabelas temporárias existe uma cláusula opcional `ON COMMIT`:

```
CREATE { GLOBAL | LOCAL } TEMPORARY TABLE nome_da_tabela ( ... ) [ ON COMMIT { DELETE
```

A cláusula `ON COMMIT` especifica se a tabela temporária deve ou não ser esvaziada quando o comando `COMMIT` é executado. Se a cláusula `ON COMMIT` for omitida, o SQL92 especifica como padrão `ON COMMIT DELETE ROWS`. Entretanto, o comportamento do PostgreSQL é sempre `ON COMMIT PRESERVE ROWS`.

“Restrição” NULL

A “restrição” `NULL` (na verdade uma não restrição) é uma extensão do PostgreSQL ao SQL92, incluída para manter a compatibilidade com alguns outros SGBDRs (e por simetria com a restrição `NOT NULL`). Sendo que este é o padrão para qualquer coluna, sua presença é simplesmente informativa.

Asserções

Uma asserção (assertion) é um tipo especial de restrição de integridade e compartilha o mesmo espaço de nomes como outras restrições. Entretanto, uma asserção não é necessriamente dependente de uma tabela em particular como as retrições são, por isso o SQL92 fornece a declaração `CREATE ASSERTION` como uma forma alternativa para definir restrições:

```
CREATE ASSERTION nome CHECK ( condição )
```

O PostgreSQL não implementa asserções atualmente.

Herança

Heranças múltiplas através da cláusula `INHERITS` é uma extensão da linguagem do PostgreSQL. O SQL99 (mas não o SQL92) define herança única utilizando uma sintaxe diferente e semânticas diferente. O estilo de herança do SQL99 ainda não é suportado pelo PostgreSQL.

Identificadores do Objeto (Object IDs)

O conceito de OIDs (identificadores de objeto) do PostgreSQL não é padrão.

Consulte também

ALTER TABLE , DROP TABLE

CREATE TABLE AS

Name

`CREATE TABLE AS` — cria uma nova tabela a partir do resultado de uma consulta

Synopsis

```
CREATE [ [ LOCAL ] { TEMPORARY | TEMP } ] TABLE nome_da_tabela [ (nome_da_coluna [, 
    AS consulta
```

Descrição

O comando `CREATE TABLE AS` cria e carrega uma tabela com dados produzidos pelo comando `SELECT`. As colunas da tabela possuem os nomes e tipos de dado associados às colunas da saída produzida pelo comando `SELECT` (exceto que é possível substituir os nomes das colunas fornecendo-se uma lista explícita de novos nomes de colunas).

O comando `CREATE TABLE AS` exibe alguma semelhança com a criação de uma visão, mas na realidade é bastante diferente: este comando cria a nova tabela e executa a consulta apenas uma vez para fazer a carga inicial dos dados da nova tabela. A nova tabela não vai ter conhecimento das próximas mudanças ocorridas na tabela de origem da consulta. Contrastando com este comportamento, uma visão executa novamente o comando `SELECT` sempre que é consultada.

Parâmetros

`[LOCAL] TEMPORARY` ou `[LOCAL] TEMP`

Se for especificado, a tabela é criada como uma tabela temporária. Tabelas temporárias são automaticamente eliminadas no final da sessão. Uma tabela persistente com o mesmo nome, caso exista, não será vista na sessão enquanto a tabela temporária existir. Todo índice criado em tabela temporária também é temporário.

A palavra `LOCAL` é opcional.

nome_da_tabela

O nome da nova tabela a ser criada. A tabela não pode existir. Entretanto, pode ser criada uma tabela temporária que possua o mesmo nome de uma tabela permanente existente.

nome_da_coluna

O nome da coluna da nova tabela. Vários nomes de colunas podem ser especificados utilizando uma lista de nomes separados por vírgula. Se os nomes das colunas não forem fornecidos, serão obtidos a partir dos nomes das colunas produzidas pela consulta.

`consulta`

Uma declaração de consulta (ou seja, um comando `SELECT`). Consulte o comando SELECT para obter a descrição da sintaxe permitida.

Diagnósticos

Consulte os comandos CREATE TABLE e SELECT para obter um sumário das possíveis mensagens de saída.

Notas

Este comando é funcionalmente equivalente ao SELECT INTO mas é preferível, porque é menos propenso a ser confundido com outros usos da sintaxe do comando `SELECT ... INTO`.

Compatibilidade

Este comando é baseado em uma funcionalidade do Oracle. Não existe nenhum comando com funcionalidade equivalente no SQL92 nem no SQL99. Entretanto, uma combinação de `CREATE TABLE` com `INSERT ... SELECT` pode ser utilizada para produzir o mesmo resultado com um pouco mais de esforço.

Histórico

O comando `CREATE TABLE AS` está disponível desde o PostgreSQL 6.3.

Consulte também

CREATE TABLE, CREATE VIEW , SELECT, SELECT INTO

CREATE TRIGGER

Name

`CREATE TRIGGER` — define um novo gatilho

Synopsis

```
CREATE TRIGGER nome { BEFORE | AFTER } { evento [OR ...] }
    ON tabela FOR EACH { ROW | STATEMENT }
    EXECUTE PROCEDURE função ( argumentos )
```

Entradas

nome

O nome do novo gatilho.

tabela

O nome de uma tabela existente.

evento

Um entre INSERT, DELETE e UPDATE.

função

Uma função fornecida pelo usuário.

Saídas

`CREATE`

Mensagem retornada se o gatilho for criado com sucesso.

Descrição

O comando `CREATE TRIGGER` introduz um novo gatilho no banco de dados atual. O gatilho fica associado com a relação *tabela* e executa a função especificada *função*.

O gatilho pode ser especificado para disparar antes (BEFORE) da operação ser realizada na tupla (antes das restrições serem verificadas e o `INSERT`, `UPDATE` ou `DELETE` serem efetuados) ou após (AFTER) a

operação ser realizada (ou seja, após as restrições serem verificadas e o `INSERT`, `UPDATE` ou `DELETE` ter completado). Se o gatilho disparar antes do evento, o gatilho pode evitar a operação para a tupla atual, ou modificar a tupla sendo inserida (para as operações de `INSERT` e `UPDATE` somente). Se o gatilho disparar após o evento todas as modificações, incluindo a última inserção, atualização ou exclusão, são "visíveis" para o gatilho.

O `SELECT` não modifica nenhuma linha, portanto não é possível criar gatilhos para `SELECT`. Regras e visões são mais apropriadas para este caso.

Consulte os capítulos sobre SPI (Interface de Programação do Servidor) e Gatilhos no *Guia do Programador do PostgreSQL* para obter mais informações.

Notas

Para criar um gatilho em uma tabela, o usuário deve possuir o privilégio `TRIGGER` na tabela.

Na versão atual, gatilhos de declaração (`STATEMENT` triggers) não estão implementados.

Consulte o comando DROP TRIGGER para obter informações sobre como remover gatilhos.

Exemplos

Verificar se o código do distribuidor existe na tabela de distribuidores antes de inserir ou atualizar uma linha da tabela filmes:

```
CREATE TRIGGER se_dist_existe
    BEFORE INSERT OR UPDATE ON filmes FOR EACH ROW
    EXECUTE PROCEDURE verificar_chave_primaria ('did', 'distribuidores', 'did');
```

Antes de remover um distribuidor, ou de atualizar o seu código, remover todas as referências para a tabela filmes:

```
CREATE TRIGGER se_filme_existe
    BEFORE DELETE OR UPDATE ON distribuidores FOR EACH ROW
    EXECUTE PROCEDURE verificar_chave_primaria (1, 'CASCADE', 'did', 'filmes', 'did')
```

O segundo exemplo também pode ser implementado usando uma chave estrangeira, como em:

```
CREATE TABLE distribuidores (
    did      DECIMAL(3),
    nome     VARCHAR(40),
    CONSTRAINT se_filme_existe
    FOREIGN KEY(did) REFERENCES filmes
    ON UPDATE CASCADE ON DELETE CASCADE
);
```

Compatibilidade

SQL92

Não existe o comando `CREATE TRIGGER` no SQL92.

SQL99

O comando `CREATE TRIGGER` do PostgreSQL implementa um subconjunto do padrão SQL99. As seguintes funcionalidades estão faltando:

- O SQL99 permite os gatilhos dispararem quando da atualização de colunas específicas (por exemplo, `AFTER UPDATE OF col1, col2`).
- O SQL99 permite definir aliás para as linhas ou tabelas "velha" e "nova" para uso na definição das ações do gatilho (por exemplo, `CREATE TRIGGER ... ON nome_tabela REFERENCING OLD ROW AS algum_nome NEW ROW AS outro_nome ...`). Como o PostgreSQL permite que os procedimentos dos gatilhos sejam escritos em qualquer linguagem definida pelo usuário, o acesso aos dados é realizado na forma específica da linguagem.
- O PostgreSQL somente possui gatilhos a nível de linha, não possuindo gatilhos a nível de declaração.
- O PostgreSQL somente permite a execução de procedimentos armazenados para a ação do gatilho. O SQL99 permite a execução de vários outros comandos SQL, como o `CREATE TABLE`, para a ação de um gatilho. Esta limitação não é difícil de ser contornada criando-se um procedimento armazenado que execute estes comandos.

Consulte também

CREATE FUNCTION, DROP TRIGGER , *Guia do Programador do PostgreSQL*

CREATE TYPE

Name

`CREATE TYPE` — define um novo tipo de dado

Synopsis

```
CREATE TYPE nome_do_tipo ( INPUT = função_de_entrada, OUTPUT = função_de_saída
      , INTERNALLENGTH = { comprimento_interno | VARIABLE }
    [ , EXTERNALLENGTH = { comprimento_externo | VARIABLE } ]
    [ , DEFAULT = padrão ]
    [ , ELEMENT = elemento ] [ , DELIMITER = delimitador ]
    [ , SEND = função_de_envio ] [ , RECEIVE = função_de_recepção ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alinhamento ]
    [ , STORAGE = armazenamento ]
)
```

Entradas

nome_do_tipo

O nome do tipo a ser criado.

comprimento_interno

Um valor literal, especificando o comprimento interno do novo tipo.

comprimento_externo

Um valor literal, especificando o comprimento externo (exibido) do novo tipo.

função_de_entrada

O nome da função criada pelo comando `CREATE FUNCTION` que converte os dados da forma externa para a forma interna do tipo.

função_de_saída

O nome da função criada pelo comando `CREATE FUNCTION` que converte os dados da forma interna numa forma adequada para ser exibida.

elemento

O tipo sendo criado é um `array`; especifica o tipo dos elementos do `array`.

delimitador

O caractere delimitador a ser usado entre os valores, nos `arrays` feitos deste tipo.

padrão

O valor padrão para o tipo de dado. Usualmente omitido, fazendo com que NULL seja o padrão.

função_de_envio

O nome da função, criada pelo comando `CREATE FUNCTION`, que converte os dados deste tipo em uma forma adequada para transmitir para outra máquina.

função_de_recepção

O nome da função, criada pelo comando `CREATE FUNCTION`, que converte os dados deste tipo da forma de transmissão para a forma interna.

alinhamento

Alinhamento de armazenamento requerido por este tipo de dado. Se for especificado, deve ser `char`, `int2`, `int4` ou `double`; o padrão é `int4`.

armazenamento

Técnica de armazenamento para este tipo de dado. Se for especificado, deve ser `plain`, `external`, `extended` ou `main`; o padrão é `plain`.

Saídas

`CREATE`

Mensagem retornada se o tipo for criado com sucesso.

Descrição

O comando `CREATE TYPE` permite ao usuário registrar um novo tipo de dado do usuário no PostgreSQL, para ser usado no banco de dados corrente. O usuário que define o tipo se torna o seu dono. O *nome_do_tipo* é o nome do novo tipo, devendo ser único entre os tipos definidos para este banco de dados.

O comando `CREATE TYPE` requer o registro de duas funções (usando CREATE FUNCTION) antes de definir o tipo. A representação do novo tipo base é determinada pela *função_de_entrada*, que converte a representação externa do tipo na representação interna utilizável pelos operadores e funções definidas para o tipo. Naturalmente a *função_de_saída* realiza a transformação inversa. A função de entrada pode ser declarada como recebendo um argumento do tipo `opaque`, ou como recebendo três argumentos dos tipos `opaque`, `OID` e `int4` (O primeiro argumento é o texto de entrada como uma cadeia de caracteres C, o segundo argumento é o tipo do elemento no caso de ser um tipo `array`, e o terceiro é o `typmod` da coluna de destino, se for conhecido). A função de saída pode ser declarada como recebendo um argumento do tipo `opaque`, ou como recebendo dois argumentos dos tipos `opaque` e `OID` (O primeiro argumento é na verdade do próprio tipo do dado, mas como a função de saída deve ser declarada primeiro, é mais fácil declará-la como recebendo o tipo `opaque`. O segundo argumento é novamente o tipo do elemento do `array` para tipos `array`).

Novos tipos de dado base podem ser de comprimento fixo e neste caso o *comprimento_interno* é um inteiro positivo, ou podem ser de comprimento variável indicado por fazer *comprimento_interno* igual a `VARIABLE` (Internamente é representado definindo `typlen` como -1). A representação interna de todos os tipos de comprimento variável devem começar por um número inteiro indicando o comprimento total deste valor do tipo.

A comprimento da representação externa é analogamente especificado usando a palavra chave *comprimento_externo* (Este valor não é utilizado atualmente, sendo tipicamente omitido, assumindo o valor padrão que é `VARIABLE`).

Para indicar que o tipo é um `array`, especifica-se o tipo dos elementos do `array` usando a palavra chave `ELEMENT`. Por exemplo, para definir um `array` de inteiros de 4 bytes ("int4"), especifica-se

```
ELEMENT = int4
```

Mais detalhes sobre os tipos `array` são mostrados abaixo.

Para indicar o delimitador a ser usado entre os valores na representação externa dos `arrays` deste tipo, o *delimitador* pode ser especificado como um caractere específico. O delimitador padrão é a vírgula, (’`,`’). Observe que o delimitador está associado com o tipo de elemento do `array`, e não com o próprio `array`.

Um valor padrão pode ser especificado, quando o usuário desejar que as colunas com este tipo de dado tenham um valor padrão diferente de NULL. Especifica-se o padrão com a palavra chave `DEFAULT` (Este tipo de padrão pode ser substituído por um cláusula `DEFAULT` explícita declarada na coluna).

Os argumentos opcionais *função_de_envio* e *função_de_recepção* não são usados atualmente, sendo usualmente omitidos (fazendo com que assumam o valor padrão que são *função_de_saída* e *função_de_entrada* respectivamente). Estas funções poderão ser ressuscitadas algum dia para especificar representações binárias independentes de máquinas.

O sinalizador opcional `PASSEDBYVALUE` indica que os valores deste tipo de dado são passados por valor, e não por referência. Observe que não se pode passar por valor os tipos cuja representação interna é maior do que o comprimento do tipo `Datum` (quatro bytes na maioria das máquinas, oito bytes em algumas).

A palavra chave *alinhamento* especifica o alinhamento do armazenamento requerido por este tipo de dado. Os valores permitidos igualam-se ao alinhamento das fronteiras de 1, 2, 4 ou 8 bytes. Observe que os tipos de tamanho variável devem ter um alinhamento de pelo menos 4, porque contêm necessariamente um `int4` como seu primeiro componente.

A palavra chave *armazenamento* permite selecionar estratégias de armazenamento para tipos de dado de comprimento variável (somente `plain` é permitido para os tipos de comprimento fixo). O `plain` desativa TOAST para o tipo de dado: será sempre armazenado em linha e não comprimido. O `extended` permite toda a capacidade do TOAST: o sistema primeiro tenta comprimir um valor longo do dado, movendo o valor para fora da tabela principal se ainda continuar muito longo. O `external` permite que o valor seja movido para fora da tabela principal, mas o sistema não vai tentar comprimi-lo. O `main` permite a compressão, mas desencoraja mover o valor para fora da tabela principal (Os itens de dado com este método de armazenamento ainda podem ser movidos para fora da tabela principal se não houver outro meio para fazer o ajuste da linha, mas será mantido na tabela principal com preferência maior que os itens `extended` e `external`).

Tipos Array

Sempre que um tipo de dado definido pelo usuário é criado, o PostgreSQL automaticamente cria um tipo `array` associado, cujo nome consiste do nome do tipo base prefixado por um caractere sublinhado. O analisador compreende esta convenção de nome e traduz as requisições para colunas do tipo `foo[]` em requisições do tipo `_foo`. O tipo `array` criado implicitamente é de comprimento variável e usa as funções de entrada e saída nativas `array_in` e `array_out`.

Pode-se perguntar com razão “porque existe a opção `ELEMENT`, se o sistema produz o tipo `array` correto automaticamente?” O único caso em que é útil utilizar `ELEMENT` é quando se constrói um tipo de comprimento fixo que é internamente um `array` de N componentes idênticos, e deseja-se que estes N componentes sejam acessados diretamente por índices em adição a qualquer outra operação que se planeje fornecer para este tipo como um todo. Por exemplo, o tipo `name` permite que seus `chars` constituintes sejam acessados desta forma. Um tipo `point` 2-D pode permitir que seus dois componentes sejam acessados como `point[0]` e `point[1]`. Observe que esta facilidade somente funciona para tipos de comprimento fixo cuja forma interna seja exatamente uma seqüência de N campos de tamanho fixo idênticos. Um tipo de comprimento variável, para permitir índices deve possuir a representação interna generalizada usada por `array_in` e `array_out`. Por razões históricas (ou seja, claramente errado mas muito tarde para mudar) os índices de `arrays` de comprimento fixo começam por zero, enquanto índices de `arrays` de comprimento variável começam por um.

Notas

Os nomes dos tipos definidos pelo usuário não podem começar pelo caractere sublinhado (“_”) e podem ter o comprimento de apenas 30 caracteres (ou, de uma maneira geral, `NAMEDATALEN-2` em vez dos `NAMEDATALEN-1` caracteres permitidos para os outros nomes). Os nomes de tipo começados por sublinhado são reservados para os nomes dos tipos `array` criados internamente.

Exemplos

Este exemplo cria o tipo de dado `caixa` e, em seguida, usa o tipo na definição de uma tabela:

```
CREATE TYPE caixa (INTERNALLENGTH = 16,
    INPUT = meu_procedimento_1, OUTPUT = meu_procedimento_2);
CREATE TABLE tbl_caixa (id INT4, descricao caixa);
```

Se a estrutura interna de `caixa` fosse um `array` contendo quatro `float4`, poderíamos escrever

```
CREATE TYPE caixa (INTERNALLENGTH = 16,
    INPUT = meu_procedimento_1, OUTPUT = meu_procedimento_2,
    ELEMENT = float4);
```

o que permitiria o valor de um componente da caixa ser acessado através do índice. Fora isso o tipo se comporta da mesma maneira que o anterior.

Este exemplo cria um tipo de objeto grande e o utiliza na definição de uma tabela:

```
CREATE TYPE objeto_grande (INPUT = lo_filein, OUTPUT = lo_fileout,
```

```
    INTERNALLENGTH = VARIABLE);
CREATE TABLE tbl_grandes_objetos (id int4, obj objeto_grande);
```

Compatibilidade

Este comando `CREATE TYPE` é uma extensão do PostgreSQL. Existe um comando `CREATE TYPE` no SQL99 que é bastante diferente nos detalhes.

Consulte também

CREATE FUNCTION, DROP TYPE , *Guia do Programador do PostgreSQL*

CREATE USER

Name

`CREATE USER` — define uma nova conta de usuário do banco de dados

Synopsis

```
CREATE USER nome_do_usuário [ [ WITH ] opção [ ... ] ]

onde opção pode ser:

   SYSID uid
        | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'senha'
        | CREATEDB | NOCREATEDB
        | CREATEUSER | NOCREATEUSER
        | IN GROUP nome_do_grupo [, ...]
        | VALID UNTIL 'data_hora'
```

Entradas

nome_do_usuário

O nome do usuário.

uid

A cláusula `SYSID` pode ser utilizada para especificar o identificador do usuário sendo criado no PostgreSQL. Não é, de forma alguma, necessário que este identificador corresponda ao identificador do usuário no UNIX, mas algumas pessoas optam por manter o mesmo número.

Se não for especificado, será utilizado por padrão o maior valor atribuído a um identificador de usuário acrescido de 1 (com mínimo de 100).

senha

Define a senha do usuário. Se não se planeja utilizar autenticação por senha pode-se omitir esta opção, mas o usuário não será capaz de se conectar a um servidor autenticado por senha. A senha pode ser definida ou alterada posteriormente através do comando *ALTER USER*.

ENCRYPTED
UNENCRYPTED

Estas palavras chave controlam se a senha é armazenada criptografada, ou não, na tabela `pg_shadow` (Se nenhuma das duas for especificada, o comportamento padrão é determinado pelo parâmetro `PASSWORD_ENCRYPTION` do servidor). Se a cadeia de caracteres apresentada já estiver criptografada no formato MD5, então esta cadeia de caracteres é armazenada como está, independentemente de ENCRYPTED ou UNENCRYPTED ser especificado. Esta funcionalidade permite a restauração das senhas criptografadas efetuadas por uma operação de “dump/restore”.

Consulte o capítulo sobre autenticação de cliente no *Guia do Administrador* para obter mais detalhes sobre como configurar os mecanismos de autenticação. Observe que os clientes antigos podem não

possuir suporte ao mecanismo de autenticação para o MD5, que é necessário para trabalhar com as senhas armazenadas criptografadas.

CREATEDB
NOCREATEDB

Estas cláusulas definem a permissão para o usuário criar bancos de dados. Se `CREATEDB` for especificado, o usuário sendo definido terá permissão para criar seus próprios bancos de dados. Se `NOCREATEDB` for especificado, nega-se ao usuário a permissão para criar bancos de dados. Se esta cláusula for omitida, `NOCREATEDB` será utilizado por padrão.

CREATEUSER
NOCREATEUSER

Estas cláusulas determinam se é permitido ou não o usuário criar novos usuários. Esta opção também torna o usuário um superusuário, que pode modificar todas as restrições de acesso. Se esta cláusula for omitida, o valor deste atributo para o usuário será `NOCREATEUSER`.

`nome_do_grupo`

O nome do grupo onde o usuário será incluído como um novo membro. Nomes de vários grupos podem estar presentes.

`data_hora`

A cláusula `VALID UNTIL` define uma data e hora após a qual a senha do usuário não é mais válida. Se esta cláusula for omitida, a conta será válida para sempre.

Saídas

`CREATE USER`

Mensagem retornada se o comando for executado com sucesso.

Descrição

O comando `CREATE USER` inclui um novo usuário em uma instância do PostgreSQL. Consulte o *Guia do Administrador* para obter mais informações sobre o gerenciamento de usuários e autenticação. Apenas os superusuários do banco de dados podem usar este comando.

Use o *ALTER USER* para mudar a senha e os privilégios do usuário, e o *DROP USER* para excluir o usuário. Use o *ALTER GROUP* para adicionar ou remover o usuário de outros grupos. O PostgreSQL possui o script *createuser* com a mesma funcionalidade deste comando (na verdade, chama este comando), mas que pode ser executado a partir da linha de comandos.

Utilização

Criar um usuário sem senha:

```
CREATE USER jonas;
```

Criar um usuário com senha:

```
CREATE USER manuel WITH PASSWORD ’jw8s0F4’;
```

Criar um usuário com senha, cuja conta seja válida até o fim de 2001. Observe que no primeiro instante de 2002 esta conta não será mais válida:

```
CREATE USER miriam WITH PASSWORD ’jw8s0F4’ VALID UNTIL ’Jan 1 2002’;
```

Criar uma conta onde o usuário pode criar bancos de dados:

```
CREATE USER manuel WITH PASSWORD ’jw8s0F4’ CREATEDB;
```

Compatibilidade

SQL92

Não existe o comando `CREATE USER` no SQL92.

CREATE VIEW

Name

`CREATE VIEW` — define uma nova visão

Synopsis

```
CREATE VIEW visão [ ( nomes_de_colunas ) ] AS SELECT consulta
```

Entradas

visão

O nome da visão a ser criada.

nomes_de_colunas

Uma relação opcional de nomes a serem usados para as colunas da visão. Quando fornecidos, estes nomes substituem os nomes inferidos a partir da consulta SQL.

consulta

Uma consulta SQL que fornece as colunas e as linhas da visão.

Consulte o comando SELECT para obter mais informações sobre os argumentos válidos.

Saídas

`CREATE`

Mensagem retornada se a visão for criada com sucesso.

`ERROR: Relation 'visão' already exists`

Este erro ocorre se a visão especificada existir no banco de dados.

`NOTICE: Attribute 'nome_da_coluna' has an unknown type`

A visão será criada possuindo uma coluna de tipo desconhecido se não for especificado. Por exemplo, o seguinte comando gera esta advertência:

```
CREATE VIEW vista AS SELECT 'Alô Mundo';
NOTICE:  Attribute '?column?' has an unknown type
        Proceeding with relation creation anyway
```

enquanto o comando abaixo não gera esta advertência:

```
CREATE VIEW vista AS SELECT text 'Alô Mundo'
```

Descrição

O comando `CREATE VIEW` define uma visão de uma tabela. A visão não é fisicamente materializada. Em vez disso, uma regra é automaticamente gerada para realizar o retorno dos dados da consulta.

Notas

Atualmente, as visões são apenas para leitura: o sistema não permite inclusão, atualização ou exclusão em uma visão. É possível obter o efeito de uma visão atualizável criando-se regras que troquem as inclusões, ... na visão por ações apropriadas em outras tabelas. Para mais informações consulte o comando CREATE RULE.

Use o comando `DROP VIEW` para excluir visões.

Utilização

Criar uma visão consistindo de todos os filmes de comédia:

```
CREATE VIEW comedias AS
    SELECT *
    FROM filmes
    WHERE tipo = ’Comédia’;

SELECT * FROM comedias;

 cod   |           titulo          | did | data_prod  |  tipo   | duracao
-------+---------------------------+-----+------------+---------+--------
 UA502 | Bananas                   | 105 | 1971-07-13 | Comédia | 01:22
 C_701 | There’s a Girl in my Soup | 107 | 1970-06-11 | Comédia | 01:36
(2 rows)
```

Compatibilidade

SQL92

O SQL92 especifica algumas funcionalidades adicionais para o comando `CREATE VIEW`:

```
CREATE VIEW visão [ coluna [, ...] ]
```

```
AS SELECT expressão [ AS nome_da_coluna ] [, ...]
FROM tabela [ WHERE condição ]
[ WITH [ CASCADE | LOCAL ] CHECK OPTION ]
```

As cláusulas opcionais para o comando SQL92 completo são:

CHECK OPTION

Esta opção está relacionada com as visões atualizáveis. Todos os comandos `INSERT` e `UPDATE` em uma visão serão verificados para garantir que os dados satisfazem as condições que definem a visão. Se não satisfizer, a atualização será rejeitada.

LOCAL

Verifica a integridade nesta visão.

CASCADE

Verifica a integridade nesta visão e em todas as visões dependentes. CASCADE é assumido se nem CASCADE nem LOCAL forem especificados.

DECLARE

Name

`DECLARE` — define um cursor

Synopsis

```
DECLARE nome_do_cursor [ BINARY ] [ INSENSITIVE ] [ SCROLL ]
    CURSOR FOR consulta
    [ FOR { READ ONLY | UPDATE [ OF coluna [, ...] ] ]
```

Entradas

nome_do_cursor

O nome do cursor a ser utilizado nas operações de busca (FETCH) posteriores.

BINARY

Faz com que o cursor busque os dados no formato binário em vez do formato texto.

INSENSITIVE

Palavra chave do SQL92 indicando que os dados buscados pelo cursor não devem ser afetados pelas atualizações feitas por outros processos ou cursores. Como as operações do cursor ocorrem dentro de uma transação no PostgreSQL, este é sempre o caso. Esta palavra chave não provoca qualquer efeito.

SCROLL

Palavra chave do SQL92 indicando que várias linhas de dados devem ser trazidas em cada operação de busca (FETCH). Como é sempre permitido pelo PostgreSQL, esta palavra chave não provoca qualquer efeito.

consulta

Uma consulta SQL que fornece as linhas a serem controladas pelo cursor. Consulte o comando SELECT para obter mais informações sobre os argumentos válidos.

READ ONLY

Palavra chave do SQL92 indicando que o cursor será usado no modo apenas para leitura. Como este é o único modo de acesso do cursor disponível no PostgreSQL, esta palavra chave não provoca qualquer efeito.

UPDATE

Palavra chave do SQL92 indicando que o cursor será usado para atualizar tabelas. Como as atualizações pelo cursor não são suportadas no momento pelo PostgreSQL, esta palavra chave provoca uma mensagem informando o erro.

`coluna`

Coluna(s) a serem atualizadas. Como as atualizações pelo cursor não são suportadas no momento pelo PostgreSQL, a cláusula UPDATE provoca uma mensagem informando o erro.

Saídas

`SELECT`

Mensagem retornada se o SELECT for executado com sucesso.

`NOTICE: Closing pre-existing portal "nome_do_cursor"`

Esta mensagem é exibida se o mesmo nome de cursor já estiver declarado no bloco de transação corrente. A definição anterior é descartada.

`ERROR: DECLARE CURSOR may only be used in begin/end transaction blocks`

Este erro ocorre quando o cursor não é declarado dentro de um bloco de transação.

Descrição

O comando `DECLARE` permite ao usuário criar cursores, os quais podem ser usados para buscar de cada vez um pequeno número de linhas de uma consulta grande. Os cursores podem retornar dados tanto no formato texto quanto binário usando o comando *FETCH*.

Os cursores normais retornam os dados no formato texto, em ASCII ou outro esquema de codificação dependendo de como o servidor PostgreSQL foi compilado. Como os dados são armazenados nativamente no formato binário, o sistema necessita fazer uma conversão para produzir o formato texto. Adicionalmente, o formato texto geralmente possui um tamanho maior do que o formato binário correspondente. Quando a informação retorna na forma de texto, o aplicativo cliente pode precisar converter para o formato binário para manipulá-la. Os cursores binários retornam os dados na representação binária nativa.

Por exemplo, se uma consulta retornar o valor “um” de uma coluna inteira, será recebida a cadeia de caracteres `1` com o cursor padrão, enquanto que o cursor binário retornaria um valor de 4 bytes igual a control-A (`^A`).

Os cursores binários devem ser usados com cuidado. Aplicativos do usuário, como o psql, não estão atentos a cursores binários, esperando que os dados cheguem no formato texto.

A representação da cadeia de caracteres é arquiteturalmente neutra, enquanto que a representação binária pode ser diferente entre máquinas com arquiteturas diferentes. *O PostgreSQL não soluciona a ordem dos bytes ou outros problemas de representação para os cursores binários*. Portanto, se a máquina cliente e a máquina servidora usam representações diferentes, (por exemplo, “big-endian” versus “little-endian”), provavelmente não vai ser desejado que os dados retornem no formato binário. Entretanto, os cursores binários podem ser um pouco mais eficientes porque existe menos sobrecarga de conversão na transferência de dados do servidor para o cliente.

Tip: Se a intenção for mostrar os dados em ASCII, buscar os dados em ASCII reduz o esforço realizado pelo cliente.

Notas

Os cursores só estão disponíveis nas transações. Use os comando *BEGIN*, *COMMIT* e *ROLLBACK* para definir um bloco de transação.

No SQL92 os cursores somente estão disponíveis nos aplicativos com SQL embutido (ESQL). O servidor PostgreSQL não implementa o comando `OPEN cursor` explícito; o cursor é considerado aberto ao ser declarado. Entretanto, o ecpg, o pré-processador SQL embutido do PostgreSQL, suporta as convenções de cursor do SQL92, incluindo as que envolvem os comandos DECLARE e OPEN.

Utilização

Para declarar um cursor:

```
DECLARE liahona CURSOR
    FOR SELECT * FROM filmes;
```

Compatibilidade

SQL92

O SQL92 permite cursores somente no SQL embutido e em módulos. O PostgreSQL permite que o cursor seja usado interativamente. O SQL92 permite aos cursores embutidos ou modulares atualizar as informações no banco de dados. Todos os cursores do PostgreSQL são apenas para leitura. A palavra chave BINARY é uma extensão do PostgreSQL.

DELETE

Name

`DELETE` — exclui linhas de uma tabela

Synopsis

```
DELETE FROM [ ONLY ] tabela [ WHERE condição ]
```

Entradas

tabela

O nome de uma tabela existente.

condição

Uma condição de consulta do SQL que retorna as linhas a serem excluídas.

Consulte o comando SELECT para obter mais informações sobre a cláusula WHERE.

Saídas

`DELETE` *contador*

Mensagem retornada se as linhas foram excluídas com sucesso. O *contador* representa o número de linhas excluídas.

Se *contador* for 0, então nenhuma linha foi excluída.

Descrição

O comando `DELETE` exclui as linhas que satisfazem a cláusula WHERE na tabela especificada.

Se a *condição* (cláusula WHERE) estiver ausente, o efeito é a exclusão de todas as linhas da tabela. O resultado vai ser uma tabela válida, porém vazia.

> **Tip:** O comando *TRUNCATE* é uma extensão do PostgreSQL que fornece um mecanismo rápido para excluir todas as linhas de uma tabela.

Por padrão o DELETE exclui as linhas da tabela especificada e de todas as suas filhas. Para atualizar somente a tabela especificada deve ser utilizada a cláusula ONLY.

É necessário possuir acesso de escrita a uma tabela para poder modificá-la, assim como acesso de leitura em todas as tabelas cujos valores são lidos na *`condição`* da cláusula WHERE.

Utilização

Remover todos os filmes, exceto os musicais:

```
DELETE FROM filmes WHERE tipo <> 'Musical';
SELECT * FROM filmes;
```

```
     cod   |           titulo          | did | data_prod  |  tipo   | duracao
-------+---------------------------+-----+------------+---------+---------
 UA501 | West Side Story           | 105 | 1961-01-03 | Musical | 02:32
 TC901 | The King and I            | 109 | 1956-08-11 | Musical | 02:13
 WD101 | Bed Knobs and Broomsticks | 111 |            | Musical | 01:57
(3 rows)
```

Esvaziar a tabela `filmes`:

```
DELETE FROM filmes;
SELECT * FROM filmes;
```

```
     cod  | titulo | did | data_prod | tipo | duracao
------+--------+-----+-----------+------+---------
(0 rows)
```

Compatibilidade

SQL92

O SQL92 permite um comando `DELETE` posicionado:

```
DELETE FROM tabela WHERE
    CURRENT OF cursor
```

onde *`cursor`* identifica um cursor aberto. Cursores interativos no PostgreSQL são apenas para leitura.

DROP AGGREGATE

Name

`DROP AGGREGATE` — remove uma função de agregação definida pelo usuário

Synopsis

```
DROP AGGREGATE nome ( tipo )
```

Entradas

nome

O nome de uma função de agregação existente.

tipo

O tipo de dado de entrada de uma função de agregação existente, ou `*` se a função aceita qualquer tipo de dado de entrada (Consulte o *Guia do Usuário do PostgreSQL* para obter mais informações sobre tipos de dado).

Saídas

`DROP`

Mensagem retornada se o comando for executado com sucesso.

`ERROR: RemoveAggregate: aggregate 'nome' for type tipo does not exist`

Esta mensagem ocorre quando a função de agregação especificada não existe no banco de dados.

Descrição

O comando `DROP AGGREGATE` remove todas as referências a uma definição de agregação existente. Para executar este comando o usuário corrente deve ser o dono da agregação.

Notas

Use *CREATE AGGREGATE* para criar funções de agregação.

Utilização

Remover a agregação `minha_media` para o tipo de dado `int4`:

```
DROP AGGREGATE minha_media(int4);
```

Compatibilidade

SQL92

Não existe o comando `DROP AGGREGATE` no SQL92; este comando é uma extensão do PostgreSQL à linguagem.

DROP DATABASE

Name

DROP DATABASE — remove um banco de dados

Synopsis

```
DROP DATABASE nome
```

Entradas

nome

O nome do banco de dados existente a ser removido

Saídas

DROP DATABASE

Mensagem retornada se o comando for executado com sucesso.

DROP DATABASE: cannot be executed on the currently open database

Não é possível estar conectado ao banco de dados que se deseja remover. Conecte-se ao banco de dados template1, ou a qualquer outro banco de dados, e execute este comando novamente.

DROP DATABASE: may not be called in a transaction block

Antes de executar este comando deve-se concluir a transação em andamento.

Descrição

O comando DROP DATABASE remove as entradas do catálogo para um banco de dados existente e remove o diretório contendo os dados. Somente pode ser executado pelo dono do banco de dados (normalmente o usuário que o criou).

O comando DROP DATABASE não pode ser desfeito. Use com prudência!

Notas

Este comando não pode ser executado enquanto conectado ao banco de dados de destino. Portanto, é mais conveniente utilizar o script *dropdb*, que é uma envoltória em torno deste comando.

Consulte o comando *CREATE DATABASE* para obter informações sobre como criar bancos de dados.

Compatibilidade

SQL92

O comando `DROP DATABASE` é uma extensão do PostgreSQL à linguagem. Não existe este comando no SQL92.

DROP FUNCTION

Name

`DROP FUNCTION` — remove uma função definida pelo usuário

Synopsis

```
DROP FUNCTION nome ( [ tipo [, ...] ] )
```

Entradas

nome

O nome de uma função existente.

tipo

O tipo de dado dos parâmetros da função.

Saídas

`DROP`

Mensagem retornada se o comando for executado com sucesso.

`NOTICE RemoveFunction: Function "nome" ("tipo") does not exist`

Esta mensagem é retornada quando a função especificada não existe no banco de dados corrente.

Descrição

O comando DROP FUNCTION remove a definição de uma função existente. Para executar este comando o usuário deve ser o dono da função. Os tipos de dado dos argumentos de entrada da função devem ser especificados, porque várias funções diferentes podem existir com o mesmo nome, mas com argumentos diferentes.

Notas

Consulte o comando CREATE FUNCTION para obter informações sobre como criar funções.

Nenhuma verificação é efetuada para garantir que os tipos, operadores, métodos de acesso, ou gatilhos que dependem desta função foram previamente removidos.

Exemplos

Este comando remove a função que calcula a raiz quadrada:

```
DROP FUNCTION sqrt(integer);
```

Compatibilidade

O comando `DROP FUNCTION` está definido no SQL99. Uma das formas da sua sintaxe é:

```
DROP FUNCTION nome (arg, ...) { RESTRICT | CASCADE }
```

onde `CASCADE` especifica a remoção de todos os objetos que dependem da função e `RESTRICT` recusa a remoção da função se existir algum objeto que dependa da função.

Consulte também

CREATE FUNCTION

DROP GROUP

Name

`DROP GROUP` — remove um grupo de usuários

Synopsis

```
DROP GROUP nome
```

Entradas

nome

O nome de um grupo existente.

Saídas

`DROP GROUP`

Mensagem retornada se o grupo for removido com sucesso.

Descrição

O comando `DROP GROUP` remove do banco de dados o grupo especificado. Os usuários cadastrados no grupo não são removidos.

Use o *CREATE GROUP* para criar grupos novos, e o *ALTER GROUP* para gerenciar os membros do grupo.

Utilização

Para eliminar um grupo:

```
DROP GROUP engenharia;
```

Compatibilidade

SQL92

Não existe o comando `DROP GROUP` no SQL92.

DROP INDEX

Name

`DROP INDEX` — remove um índice

Synopsis

```
DROP INDEX nome_do_índice [, ...]
```

Entradas

nome_do_índice

O nome do índice a ser removido.

Saídas

`DROP`

Mensagem retornada se o comando for executado com sucesso.

`ERROR: index "nome_do_índice" does not exist`

Esta mensagem ocorre se *nome_do_índice* não for um índice no banco de dados.

Descrição

O comando `DROP INDEX` remove um índice existente no sistema de banco de dados. Para executar este comando é necessário ser o dono do índice.

Notas

O comando `DROP INDEX` é uma extensão da linguagem do PostgreSQL.

Consulte o comando *CREATE INDEX* para obter informações sobre como criar índices.

Utilização

Remover o índice `titulo_idx`:

```
DROP INDEX titulo_idx;
```

Compatibilidade

SQL92

O SQL92 define comandos através dos quais um banco de dados relacional genérico pode ser acessado. O índice é uma funcionalidade dependente da implementação e portanto não existe comando específico para índice ou a sua definição na linguagem SQL92.

DROP LANGUAGE

Name

`DROP LANGUAGE` — remove uma linguagem procedural definida pelo usuário

Synopsis

```
DROP [ PROCEDURAL ] LANGUAGE nome
```

Entradas

nome

O nome de uma linguagem procedural existente. Para manter a compatibilidade com as versões anteriores o nome pode estar entre apóstrofos (`'`).

Saídas

`DROP`

Mensagem retornada se a linguagem for removida com sucesso.

`ERROR: Language "nome" doesn't exist`

Esta mensagem ocorre quando a linguagem chamada *nome* não é encontrada no banco de dados.

Descrição

O comando `DROP PROCEDURAL LANGUAGE` remove a definição da linguagem procedural registrada anteriormente chamada *nome*.

Notas

O comando `DROP PROCEDURAL LANGUAGE` é uma extensão do PostgreSQL à linguagem.

Consulte o comando *CREATE LANGUAGE* para obter informações sobre como criar linguagens procedurais.

Nao é verificado se existem funções ou procedimentos de gatilhos registrados nesta linguagem. Para habilitá-los novamente sem ter que remover e recriar todas as funções, o atributo prolang do pg_proc das

funções deve ser ajustado para o novo ID do objeto da entrada da pg_language recriada para a linguagem procedural.

Utilização

Remover a linguagem PL/Sample:

```
DROP LANGUAGE plsample;
```

Compatibilidade

SQL92

Não existe o comando `DROP PROCEDURAL LANGUAGE` no SQL92.

DROP OPERATOR

Name

`DROP OPERATOR` — remove um operador definido pelo usuário

Synopsis

```
DROP OPERATOR id ( tipo_esquerdo | NONE , tipo_direito | NONE )
```

Entradas

id

O identificador de um operador existente.

tipo_esquerdo

O tipo do argumento do lado esquerdo do operador; escreva `NONE` se o operador não possuir argumento do lado esquerdo.

tipo_direito

O tipo do argumento do lado direito do operador; escreva `NONE` se o operador não possuir argumento do lado direito.

Saídas

`DROP`

Mensagem retornada se o comando for executado com sucesso.

`ERROR: RemoveOperator: binary operator 'operador' taking 'tipo_esquerdo' and 'tipo_direito' does not exist`

Esta mensagem ocorre quando o operador binário especificado não existe.

`ERROR: RemoveOperator: left unary operator 'opererador' taking 'tipo_esquerdo' does not exist`

Esta mensagem ocorre quando o operador unário esquerdo especificado não existe.

`ERROR: RemoveOperator: right unary operator 'operador' taking 'tipo_direito' does not exist`

Esta mensagem ocorre quando o operador unário direito especificado não existe.

Descrição

O comando `DROP OPERATOR` remove um operador existente do banco de dados. Para executar este comando é necessário ser o dono do operador.

O tipo do lado esquerdo ou direito de um operador unário esquerdo ou direito, respectivamente, deve ser especificado como `NONE`.

Notas

O comando `DROP OPERATOR` é uma extensão da linguagem do PostgreSQL.

Consulte o comando *CREATE OPERATOR* para obter informações sobre como criar operadores.

É responsabilidade do usuário remover todos os métodos de acesso e classes de operadores que dependam do operador removido.

Utilização

Remover o operador de potência `a^n` para `int4`:

```
DROP OPERATOR ^ (int4, int4);
```

Remover o operador de negação unário esquerdo (`! b`) para `booleano`:

```
DROP OPERATOR ! (none, bool);
```

Remover o operador de fatorial unário direito (`i !`) para `int4`:

```
DROP OPERATOR ! (int4, none);
```

Compatibilidade

SQL92

Não existe o comando `DROP OPERATOR` no SQL92.

DROP RULE

Name

`DROP RULE` — remove uma regra

Synopsis

```
DROP RULE nome [, ...]
```

Entradas

nome

O nome de uma regra existente a ser removida.

Saídas

`DROP`

Mensagem retornada se o comando for executado com sucesso.

`ERROR: Rule or view "nome" not found`

Esta mensagem ocorre quando a regra especificada não existe.

Descrição

O comando `DROP RULE` remove uma regra do sistema de regras do PostgreSQL. O PostgreSQL cessa imediatamente de impô-la e remove sua definição dos catálogos do sistema.

Notas

O comando `DROP RULE` é uma extensão do PostgreSQL à linguagem.

Consulte o comando `CREATE RULE` para obter informações sobre como criar regras.

Quando a regra é removida, o acesso às informações históricas escritas pela regra pode desaparecer.

Utilização

Para remover a regra `nova_regra`:

```
DROP RULE nova_regra;
```

Compatibilidade

SQL92

Não existe o comando `DROP RULE` no SQL92.

DROP SEQUENCE

Name

DROP SEQUENCE — remove uma seqüência

Synopsis

```
DROP SEQUENCE nome [, ...]
```

Entradas

nome

O nome da seqüência.

Saídas

DROP

Mensagem retornada se a seqüência for removida com sucesso.

ERROR: sequence "*nome*" does not exist

Esta mensagem ocorre quando a seqüência especificada não existe.

Descrição

O comando DROP SEQUENCE remove do banco de dados o gerador de números seqüenciais. Na implementação atual das seqüências como tabelas especiais, este comando funciona de maneira análoga ao comando DROP TABLE.

Notas

O comando DROP SEQUENCE é uma extensão do PostgreSQL à linguagem.

Consulte o comando CREATE SEQUENCE para obter informações sobre como criar seqüências.

Utilização

Para remover do banco de dados a seqüência `serial`:

```
DROP SEQUENCE serial;
```

Compatibilidade

SQL92

Não existe o comando `DROP SEQUENCE` no SQL92.

DROP TABLE

Name

`DROP TABLE` — remove uma tabela

Synopsis

```
DROP TABLE nome [, ...]
```

Entradas

nome

O nome de uma tabela existente a ser removida.

Saídas

`DROP`

Mensagem retornada se o comando for executado com sucesso.

`ERROR: table "nome" does not exist`

Se a tabela especificada não existe no banco de dados.

Descrição

O comando `DROP TABLE` remove tabelas do banco de dados. Somente o criador pode remover a tabela. A tabela poderá ficar sem linhas, mas não será removida, usando o comando `DELETE`.

Se a tabela sendo destruída possuir índices secundários nela, estes índices serão removidos primeiro. A remoção apenas do índice secundário não afeta o conteúdo da tabela subjacente.

Notas

Consulte os comandos `CREATE TABLE` e `ALTER TABLE` para obter informações sobre como criar e modificar tabelas.

Utilização

Destruir as tabelas `filmes` e `distribuidores`:

```
DROP TABLE filmes, distribuidores;
```

Compatibilidade

SQL92

O SQL92 especifica algumas funcionalidades adicionais para o comando DROP TABLE:

```
DROP TABLE tabela { RESTRICT | CASCADE }
```

RESTRICT

Garante que somente uma tabela sem visões dependentes ou restrições de integridade pode ser destruída.

CASCADE

Toda visão ou restrição de integridade que faça referência à tabela também será removida.

Tip: Atualmente, as visões que fazem referência à tabela devem ser removidas explicitamente.

DROP TRIGGER

Name

`DROP TRIGGER` — remove um gatilho

Synopsis

```
DROP TRIGGER nome ON tabela
```

Entradas

nome

O nome de um gatilho existente.

tabela

O nome da tabela.

Saídas

`DROP`

Mensagem retornada se o gatilho for removido com sucesso.

`ERROR: DropTrigger: there is no trigger nome on relation "tabela"`

Esta mensagem ocorre se o gatilho especificado não existe.

Descrição

O comando `DROP TRIGGER` remove uma definição de gatilho existente. Para executar este comando o usuário corrente deve ser o dono da tabela para a qual o gatilho está definido.

Exemplos

Remover o gatilho `se_dist_existe` da tabela `filmes`:

```
DROP TRIGGER se_dist_existe ON filmes;
```

Compatibilidade

SQL92

Não existe o comando `DROP TRIGGER` no SQL92.

SQL99

O comando `DROP TRIGGER` do PostgreSQL não é compatível com o SQL99. No SQL99 os nomes dos gatilhos não são locais às tabelas, portanto o comando é simplesmente `DROP TRIGGER` *`nome`*.

Consulte também

CREATE TRIGGER

DROP TYPE

Name

`DROP TYPE` — remove um tipo de dado definido pelo usuário

Synopsis

```
DROP TYPE nome_do_tipo [, ...]
```

Entradas

nome_do_tipo

O nome de um tipo existente.

Saídas

`DROP`

Mensagem retornada se o comando for executado com sucesso.

`ERROR: RemoveType: type 'nome_do_tipo' does not exist`

Esta mensagem ocorre quando o tipo especificado não é encontrado.

Descrição

O comando `DROP TYPE` remove um tipo do usuário dos catálogos do sistema.

Somente o dono do tipo pode removê-lo.

Notas

- É responsabilidade do usuário remover os operadores, funções, agregações, métodos de acesso, subtipos e tabelas que usam o tipo removido. Entretanto, o tipo de dado `array` associado (que é automaticamente criado pelo comando `CREATE TYPE`) é removido automaticamente.
- Se um tipo nativo do PostgreSQL for removido, o comportamento do servidor torna-se imprevisível.

Exemplos

Para remover o tipo `caixa`:

```
DROP TYPE caixa;
```

Compatibilidade

O comando `DROP TYPE` existe no SQL99. Assim como a maioria dos outros comandos “drop”, o `DROP TYPE` do SQL99 requer uma cláusula de “comportamento da remoção”, para selecionar entre remover todos os objetos dependentes ou não remover se existir algum objeto dependente:

```
DROP TYPE nome { CASCADE | RESTRICT }
```

O PostgreSQL atualmente ignora inteiramente as dependências.

Observe que o comando `CREATE TYPE` e os mecanismos de extensão de tipo do PostgreSQL são diferentes do SQL99.

Consulte também

CREATE TYPE

DROP USER

Name

`DROP USER` — remove uma conta de usuário do banco de dados

Synopsis

```
DROP USER nome
```

Entradas

nome

O nome de um usuário existente.

Saídas

`DROP USER`

Mensagem retornada se o usuário for removido com sucesso.

`ERROR: DROP USER: user "nome" does not exist`

Esta mensagem ocorre quando o nome do usuário não é encontrado.

`DROP USER: user "nome" owns database "nome", cannot be removed`

Deve-se primeiro remover o banco de dados ou mudar seu dono.

Descrição

O comando `DROP USER` remove o usuário especificado do banco de dados. Não remove tabelas, visões ou outros objetos de propriedade do usuário. Se o usuário possuir algum banco de dados uma mensagem de erro é gerada.

Use o comando *CREATE USER* para adicionar novos usuários, e o *ALTER USER* para mudar seus atributos. O PostgreSQL tem o aplicativo *dropuser* que possui a mesma funcionalidade deste comando (na verdade, chama este comando) mas que pode ser executado a partir da linha de comandos.

Utilização

Para remover uma conta de usuário:

```
DROP USER josias;
```

Compatibilidade

SQL92

Não existe o comando `DROP USER` no SQL92.

DROP VIEW

Name

`DROP VIEW` — remove uma visão

Synopsis

```
DROP VIEW nome [, ...]
```

Entradas

nome

O nome de uma visão existente.

Saídas

`DROP`

Mensagem retornada se o comando for executado com sucesso.

`ERROR: view nome does not exist`

Esta mensagem ocorre quando a visão especificada não existe no banco de dados.

Descrição

O comando `DROP VIEW` remove do banco de dados uma visão existente. Para executar este comando é necessário ser o dono da visão.

Notas

Consulte o comando `CREATE VIEW` para obter informações sobre como criar visões.

Utilização

Remover a visão chamada `tipos`:

```
DROP VIEW tipos;
```

Compatibilidade

SQL92

O SQL92 especifica algumas funcionalidades adicionais para o comando `DROP VIEW`:

```
DROP VIEW visão { RESTRICT | CASCADE }
```

Entradas

RESTRICT

Garante que somente uma visão sem restrições de integridade nem visões dependentes pode ser removida.

CASCADE

Toda visão que faça referência e restrição de integridade também é removida.

Notas

Atualmente, a remoção de uma visão referenciada do banco de dados PostgreSQL deve ser feita explicitamente.

END

Name

`END` — efetiva a transação corrente

Synopsis

```
END [ WORK | TRANSACTION ]
```

Entradas

WORK
TRANSACTION

Palavras chave opcionais. Não produzem nenhum efeito.

Saídas

`COMMIT`

Mensagem retornada se a transação for efetivada com sucesso.

`NOTICE: COMMIT: no transaction in progress`

Se não houver nenhuma transação sendo executada.

Descrição

O comando `END` é uma extensão do PostgreSQL. É um sinônimo para o comando *COMMIT* compatível com o SQL92.

Notas

As palavras chave WORK e TRANSACTION são informativas podendo ser omitidas.

Use o *ROLLBACK* para desfazer a transação.

Utilização

Para tornar todas as modificações permanentes:

```
END WORK;
```

Compatibilidade

SQL92

O comando `END` é uma extensão do PostgreSQL que fornece uma funcionalidade equivalente ao *COMMIT*.

EXPLAIN

Name

`EXPLAIN` — mostra o plano de execução de uma instrução

Synopsis

```
EXPLAIN [ ANALYZE ] [ VERBOSE ] consulta
```

Entradas

ANALYZE

Sinalizador para executar a consulta e mostrar os tempos reais de execução.

VERBOSE

Sinalizador para mostrar o plano de execução detalhado.

`consulta`

Qualquer *`consulta`*.

Saídas

`NOTICE: QUERY PLAN:` *`plano`*

Plano de execução explícito do servidor PostgreSQL.

`EXPLAIN`

Sinalizador enviado após o plano da consulta ter sido mostrado.

Descrição

Este comando mostra o plano de execução que o planejador do PostgreSQL gera para a consulta fornecida. O plano de execução mostra como a(s) tabela(s) referenciada pela consulta será varrida --- por uma varredura seqüencial, varredura do índice, etc. --- e, se várias tabelas forem referenciadas, quais algoritmos de junção serão usados para unir as tuplas requisitadas de cada tabela de entrada.

A parte mais crítica mostrada é o custo estimado da execução da consulta, que é a estimativa feita pelo planejador sobre a duração da execução da consulta (medida em unidade de acesso às páginas do disco).

Na verdade dois números são mostrados: o tempo inicial anterior à primeira tupla poder ser retornada, e o tempo total para retornar todas as tuplas. Para a maioria das consultas o tempo total é o que interessa, mas em contextos como os das subconsultas EXISTS o planejador escolhe o menor tempo inicial em vez do menor tempo total (porque o executor pára após ter obtido uma linha). Além disso, se for limitado o número de tuplas a serem retornadas usando a cláusula LIMIT, o planejador efetua uma interpolação apropriada com relação aos custos finais para saber qual plano é realmente o de menor custo.

A opção ANALYZE faz com que a consulta seja realmente executada, e não apenas planejada. O tempo total de duração gasto dentro de cada parte do plano (em milissegundos) e o número total de linhas realmente retornadas são adicionados ao que é normalmente mostrado. Esta opção é útil para ver se as estimativas do planejador estão próximas da realidade.

A opção VERBOSE fornece a representação interna completa da árvore do plano, em vez de apenas o seu sumário (e a envia para o arquivo de `log` do `postmaster`, também). Geralmente esta opção é útil apenas para fazer o `debug` do próprio PostgreSQL.

Caution

Tenha em mente que a consulta é realmente executada quando a opção ANALYZE é usada. Embora o `EXPLAIN` despreze qualquer saída que o SELECT possa retornar, outros efeitos colaterais da consulta acontecem na forma usual. Para usar `EXPLAIN ANALYZE` em um INSERT, UPDATE, ou DELETE sem deixar que os dados sejam afetados, utilize o seguinte procedimento:

```
BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

Notas

Existem apenas informações esparsas sobre a utilização do otimizador na documentação do PostgreSQL. Consulte o *Guia do Usuário* e o *Guia do Programador* para obter mais informações.

Utilização

Para mostrar o plano da consulta para uma consulta simples em uma tabela com uma única coluna `int4` e 128 linhas:

```
    EXPLAIN SELECT * FROM foo;
        NOTICE:  QUERY PLAN:

Seq Scan on foo  (cost=0.00..2.28 rows=128 width=4)

EXPLAIN
```

Para a mesma tabela contendo um índice para dar suporte a uma condição de *eqüijunção* na consulta, o comando `EXPLAIN` mostra um plano diferente:

```
    EXPLAIN SELECT * FROM foo WHERE i = 4;
        NOTICE:  QUERY PLAN:

Index Scan using fi on foo  (cost=0.00..0.42 rows=1 width=4)

EXPLAIN
```

E finalmente, para a mesma tabela contendo um índice para dar suporte a uma condição de *eqüijunção* na consulta, o comando `EXPLAIN` mostra o seguinte para uma consulta contendo uma função de agregação:

```
    EXPLAIN SELECT sum(i) FROM foo WHERE i = 4;
        NOTICE:  QUERY PLAN:

Aggregate  (cost=0.42..0.42 rows=1 width=4)
  ->  Index Scan using fi on foo  (cost=0.00..0.42 rows=1 width=4)
```

Observe que os números específicos mostrados, e mesmo a estratégia selecionada para a consulta, pode variar entre versões do PostgreSQL devido a melhorias no planejador.

Compatibilidade

SQL92

Não existe o comando `EXPLAIN` no SQL92.

FETCH

Name

`FETCH` — busca linhas de uma tabela usando um cursor

Synopsis

```
FETCH [ direção ] [ contador ] { IN | FROM } cursor
FETCH [ FORWARD | BACKWARD | RELATIVE ] [ # | ALL | NEXT | PRIOR ] { IN | FROM } curs
```

Entradas

direção

Um *seletor* definindo a direção da busca. Pode ser um dos seguintes:

FORWARD

busca a(s) próxima(s) linha(s). Este é o padrão quando o *seletor* for omitido.

BACKWARD

busca a(s) linha(s) anterior(es).

RELATIVE

Somente por compatibilidade com o SQL92.

contador

O *contador* determina quantas linhas serão buscadas. Pode ser um dos seguintes:

#

Um número inteiro com sinal que especifica quantas linhas serão buscadas. Observe que um número negativo é equivalente a mudar o sentido de FORWARD e de BACKWARD.

ALL

Busca todas as linhas remanescentes.

NEXT

Equivalente a especificar um contador igual a `1`.

PRIOR

Equivalente a especificar um contador igual a `-1`.

cursor

O nome de um cursor aberto.

Saídas

O comando `FETCH` retorna o resultado da consulta definida pelo cursor especificado. As seguintes mensagens retornam quando a consulta falha:

`NOTICE: PerformPortalFetch: portal "cursor" not found`

Se o *cursor* não tiver sido previamente declarado. O cursor deve ser declarado dentro do bloco da transação.

`NOTICE: FETCH/ABSOLUTE not supported, using RELATIVE`

O PostgreSQL não suporta o posicionamento absoluto do cursor.

`ERROR: FETCH/RELATIVE at current position is not supported`

O SQL92 permite buscar repetitivamente o cursor em sua “posição corrente” usando a sintaxe

```
FETCH RELATIVE 0 FROM cursor.
```

O PostgreSQL atualmente não suporta esta noção; na verdade o valor zero é reservado para indicar que todas as linhas devem ser buscadas, equivalendo a especificar a palavra chave ALL. Se a palavra chave RELATIVE for usada, o PostgreSQL assume que o usuário deseja o comportamento do SQL92 e retorna esta mensagem de erro.

Descrição

O comando `FETCH` permite o usuário buscar linhas usando um cursor. O número de linhas buscadas é especificado pelo #. Se o número de linhas remanescentes no cursor for menor do que #, então somente as linhas disponíveis são buscadas. Colocando-se a palavra chave ALL no lugar do número faz com que todas as linhas restantes no cursor sejam buscadas. As instâncias podem ser buscadas para frente (FORWARD) e para trás (BACKWARD) O padrão é para frente.

> **Tip:** É permitido especificar números negativos para o contador de linhas. Um número negativo equivale a inverter o sentido das palavras chave FORWARD e BACKWARD. Por exemplo, `FORWARD -1` é o mesmo que `BACKWARD 1`.

Notas

Observe que as palavras chave FORWARD e BACKWARD são extensões do PostgreSQL. A sintaxe do SQL92 também é suportada, especificando-se a segunda forma do comando. Veja abaixo para obter detalhes sobre questões de compatibilidade.

A atualização de dados pelo cursor não é suportada pelo PostgreSQL, porque mapear as atualizações do cursor de volta para as tabelas base geralmente não é possível, como no caso das atualizações nas visões. Conseqüentemente, os usuários devem executar um comando UPDATE explícito para atualizar os dados.

Os cursores somente podem ser usados dentro de transações, porque os dados por eles armazenados estendem-se por muitas consultas dos usuários.

Use o comando *MOVE* para mudar a posição do cursor. O comando *DECLARE* define um cursor. Consulte os comando *BEGIN*, *COMMIT*, e *ROLLBACK* para obter mais detalhes sobre transações.

Utilização

O exemplo a seguir acessa uma tabela usando um cursor.

```
-- Definir e usar o cursor:

BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM filmes;

-- Buscar as 5 primeiras linhas do cursor liahona:
FETCH FORWARD 5 IN liahona;
```

```
  cod   |          titulo         | did | data_prod  |  tipo    | duracao
-------+-------------------------+-----+------------+----------+---------
 BL101 | The Third Man           | 101 | 1949-12-23 | Drama    | 01:44
 BL102 | The African Queen       | 101 | 1951-08-11 | Romance  | 01:43
 JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romance  | 01:25
 P_301 | Vertigo                 | 103 | 1958-11-14 | Ação     | 02:08
 P_302 | Becket                  | 103 | 1964-02-03 | Drama    | 02:28
```

```
-- Buscar a linha anterior:
FETCH BACKWARD 1 IN liahona;
```

```
  cod   | titulo  | did | data_prod  | tipo   | duracao
-------+---------+-----+------------+--------+---------
 P_301 | Vertigo | 103 | 1958-11-14 | Ação   | 02:08
```

```
-- fechar a consulta e efetivar o trabalho:

CLOSE liahona;
COMMIT WORK;
```

Compatibilidade

SQL92

> **Note:** O uso não embutido de cursores é uma extensão do PostgreSQL. A sintaxe e a utilização dos cursores está sendo comparada com relação à forma embutida dos cursores definida no SQL92.

O SQL92 permite o posicionamento absoluto de cursores para a busca (FETCH), e permite armazenar os resultados em variáveis explicitadas:

```
FETCH ABSOLUTE #
    FROM cursor
    INTO :variável [, ...]
```

ABSOLUTE

O cursor deve ser posicionado na linha especificada pelo número absoluto. Todos os números de linhas no PostgreSQL são números relativos, portanto esta funcionalidade não é suportada.

:*variável*

Variável do programa.

GRANT

Name

GRANT — define privilégios de acesso

Synopsis

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIGGER } [,...] |
    ON [ TABLE ] nome_do_objeto [, ...]
    TO { nome_do_usuário | GROUP nome_do_grupo | PUBLIC } [, ...]
```

Descrição

O comando `GRANT` concede permissões específicas no objeto (tabela, visão, seqüência) para um ou mais usuários ou grupos de usuário. Estas permissões são adicionadas às já concedidas, caso existam.

A palavra chave `PUBLIC` indica que o privilégio deve ser concedido para todos os usuários, inclusive aos que vierem a ser criado posteriormente. `PUBLIC` pode ser considerado como um grupo implicitamente definido que sempre inclui todos os usuários. Observe que qualquer usuário em particular possui a soma dos privilégios concedidos diretamente para o mesmo, mais os privilégios concedidos para qualquer grupo do qual seja membro, mais os privilégios concedidos para `PUBLIC`.

Os usuários, fora o criador do objeto, não possuem nenhum privilégio de acesso ao objeto a menos que o criador conceda as permissões. Não existe nenhuma necessidade de se conceder privilégios ao criador do objeto, porque o criador automaticamente possui todos os privilégios (O criador pode, entretanto, decidir revogar alguns de seus próprios privilégios por motivo de segurança. Observe que esta capacidade de conceder e revogar privilégios é inerente ao criador, não podendo ser perdida. O direito de eliminar o objeto é da mesma forma inerente ao criador, não podendo ser concedido ou revogado).

Os privilégios possíveis são:

SELECT

Permite consultar qualquer coluna (*SELECT*) da tabela, visão ou seqüência especificada. Também permite utilizar o comando *COPY* FROM.

INSERT

Permite incluir novas linhas (*INSERT*) na tabela especificada. Também permite utilizar o comando *COPY* TO.

UPDATE

Permite modificar os dados de qualquer coluna (*UPDATE*) da tabela especificada. O comando `SELECT ... FOR UPDATE` também requer este privilégio (além do privilégio `SELECT`). Para as seqüências, este privilégio permite o uso de `nextval`, `currval` e `setval`.

DELETE

Permite excluir linhas (*DELETE*) da tabela especificada.

RULE

Permite criar regras para a tabela ou para a visão (Consulte o comando *CREATE RULE*).

REFERENCES

Para criar uma tabela com restrição de chave estrangeira é necessário possuir este privilégio na tabela com a chave referenciada.

TRIGGER

Permite criar gatilhos na tabela especificada (Consulte o comando *CREATE TRIGGER*).

ALL PRIVILEGES

Concede todos os privilégios mostrados acima de uma só vez. A palavra chave `PRIVILEGES` é opcional no PostgreSQL, entretanto é requerida pelo SQL estrito.

Os privilégios requeridos pelos outros comandos estão listados nas páginas de referência dos respectivos comandos.

Notas

Deve-se notar que os *superusuários* do banco de dados podem acessar todos os objetos a despeito dos privilégios definidos. Este comportamento é comparável aos direitos do usuário `root` no sistema Unix. Assim como o `root`, não é aconselhável operar como um superusuário, exceto quando for absolutamente necessário.

Atualmente para conceder privilégios somente para algumas colunas deve-se criar uma visão possuindo as colunas desejadas e conceder os privilégios para esta visão.

Use o comando `\z` do psql para obter informações sobre os privilégios concedidos nos objetos existentes:

```
       Database    = lusitania
+------------------+---------------------------------------------+
|  Relation        |        Grant/Revoke Permissions             |
+------------------+---------------------------------------------+
| mytable          | {"=rw","miriam=arwdRxt","group todos=rw"}   |
+------------------+---------------------------------------------+
Legend:
      uname=arwR -- privileges granted to a user
group gname=arwR -- privileges granted to a group
           =arwR -- privileges granted to PUBLIC

               r -- SELECT ("read")
               w -- UPDATE ("write")
               a -- INSERT ("append")
               d -- DELETE
               R -- RULE
               x -- REFERENCES
               t -- TRIGGER
         arwdRxt -- ALL PRIVILEGES
```

O comando *REVOKE* é utilizado para revogar os privilégios de acesso.

Exemplos

Conceder, para todos os usuários, o privilégio de inserir na tabela filmes:

```
GRANT INSERT ON filmes TO PUBLIC;
```

Conceder todos os privilégios ao usuário `manuel` na visão `tipos`:

```
GRANT ALL PRIVILEGES ON tipos TO manuel;
```

Compatibilidade

SQL92

A palavra chave `PRIVILEGES` em `ALL PRIVILEGES` é requerida. O SQL não aceita a concessão de privilégios em mais de uma tabela no mesmo comando.

A sintaxe do SQL92 para o comando GRANT permite conceder privilégios individuais para as colunas da tabela, e permite conceder o privilégio de conceder o mesmo privilégio para outros:

```
GRANT privilégio [, ...]
    ON objeto [ ( coluna [, ...] ) ] [, ...]
    TO { PUBLIC | nome_do_usuário [, ...] } [ WITH GRANT OPTION ]
```

O SQL permite conceder o privilégio USAGE em outros tipos de objeto: CHARACTER SET, COLLATION, TRANSLATION, DOMAIN.

O privilégio TRIGGER foi introduzido no SQL99. O privilégio RULE é uma extensão do PostgreSQL.

Consulte também

REVOKE

INSERT

Name

`INSERT` — cria novas linhas na tabela

Synopsis

```
INSERT INTO tabela [ ( coluna [, ...] ) ]
    { DEFAULT VALUES | VALUES ( expressão [, ...] ) | SELECT consulta }
```

Entradas

tabela

O nome de uma tabela existente.

coluna

O nome de uma coluna da *tabela*.

DEFAULT VALUES

Todas as colunas são preenchidas com nulo, ou pelos valores especificados quando a tabela é criada usando a cláusula DEFAULT.

expressão

Uma expressão ou valor válido a ser atribuído à *coluna*.

consulta

Uma consulta válida. Consulte o comando SELECT para obter uma descrição mais detalhada dos argumentos válidos.

Saídas

`INSERT` *oid* `1`

Mensagem retornada se somente uma linha for incluída. O identificador do objeto *oid* é o OID numérico da linha incluída.

`INSERT 0` *#*

Mensagem retornada se mais de uma linha for incluída. O # é o número de linhas incluídas.

Descrição

O comando `INSERT` permite a inclusão de novas linhas na tabela. Pode ser incluída uma linha de cada vez, ou várias linhas como o resultado de uma consulta. As colunas da lista de inserção podem estar em qualquer ordem.

As colunas que não aparecem na lista de inserção são incluídas utilizando o valor padrão, seja o valor DEFAULT declarado, ou nulo. O PostgreSQL rejeita a nova coluna se um valor nulo for incluído em uma coluna declarada como NOT NULL.

Se a expressão para cada coluna não for do tipo de dado correto, será tentada uma conversão automática de tipo.

É necessário possuir o privilégio INSERT na tabela para inserir linhas, assim como o privilégio SELECT nas tabelas especificadas na cláusula WHERE.

Utilização

Inserir uma única linha na tabela `filmes`:

```
INSERT INTO filmes VALUES
    (’UA502’,’Bananas’,105,’1971-07-13’,’Comédia’,INTERVAL ’82 minute’);
```

No segundo exemplo, mostrado abaixo, a última coluna `duração` foi omitida e, portanto, vai receber o valor padrão NULL:

```
INSERT INTO filmes (codigo, titulo, did, data_prod, tipo)
    VALUES (’T_601’, ’Yojimbo’, 106, DATE ’1961-06-16’, ’Drama’);
```

Inserir uma única linha na tabela distribuidores; observe que somente a coluna `nome` está especificada, portanto vai ser atribuído o valor padrão para a coluna `did` que foi omitida:

```
INSERT INTO distribuidores (nome) VALUES (’Entrega Rápida’);
```

Inserir várias linhas na tabela filmes a partir da tabela `tmp`:

```
INSERT INTO filmes SELECT * FROM tmp;
```

Inclusão em `arrays` (Consulte o *Guia do Usuário do PostgreSQL* para obter mais informações sobre `arrays`):

```
-- Criar um tabuleiro vazio de 3x3 posições para o Jogo da Velha
-- (todos 3 comandos criam o mesmo atributo do tabuleiro)
```

```
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1,'{{"","",""},{},{"",""}}');
INSERT INTO tictactoe (game, board[3][3])
    VALUES (2,'{}');
INSERT INTO tictactoe (game, board)
    VALUES (3,'{{„},{„},{„}}');
```

Compatibilidade

SQL92

O comando `INSERT` é totalmente compatível com o SQL92. As possíveis limitações nas funcionalidades da cláusula de *`consulta`* estão documentadas no comando *SELECT*.

LISTEN

Name

`LISTEN` — escuta uma notificação

Synopsis

```
LISTEN nome
```

Entradas

nome

O nome da condição de notificação.

Saída

`LISTEN`

Mensagem retornada se a notificação for registrada com sucesso.

`NOTICE Async_Listen: We are already listening on` *nome*

Este processo servidor já está registrado para esta condição de notificação.

Descrição

O comando `LISTEN` registra o processo servidor corrente do PostgreSQL para escutar a condição de notificação *nome*.

Sempre que o comando `NOTIFY` *nome* é executado, tanto por este processo servidor quanto por qualquer outro conectado ao mesmo banco de dados, todos os processos servidores escutando esta condição de notificação neste instante são notificados, e cada um por sua vez notifica o aplicativo cliente ao qual está conectado. Consulte a discussão do comando `NOTIFY` para obter mais informações.

Um processo servidor pode cancelar seu registro para uma dada condição de notificação usando o comando `UNLISTEN`. Os registros de escuta de um processo servidor são automaticamente removidos quando o processo servidor encerra sua execução.

O método que o aplicativo cliente deve usar para detectar eventos de notificação depende da interface de programação de aplicativos (API) do PostgreSQL sendo usada. Usando a biblioteca libpq básica, o aplicativo executa o comando `LISTEN` como um comando SQL comum e depois passa a chamar periodicamente a rotina `PQnotifies` para descobrir se algum evento de notificação foi recebido. Outras interfaces, como a libpgtcl, fornecem métodos de nível mais alto para tratar os eventos de notificação; na verdade, usando a libpgtcl o programador do aplicativo não precisa nem executar o comando `LISTEN` ou `UNLISTEN` diretamente. Consulte a documentação da biblioteca sendo usada para obter mais detalhes.

O comando *NOTIFY* contém uma discussão mais extensa da utilização do comando `LISTEN` e do comando `NOTIFY`.

Notas

`nome` pode ser qualquer cadeia de caracteres válida como um nome; não precisa corresponder ao nome de qualquer tabela existente. Se *`nome_notificação`* estiver entre aspas ("), não é necessário nem que seja um nome com uma sintaxe válida; pode ser qualquer cadeia de caracteres com até 31 caracteres.

Em algumas versões anteriores do PostgreSQL, o *`nome`* tinha que vir entre aspas quando não correspondia a nenhum nome de tabela existente, mesmo que sintaticamente fosse um nome válido, mas não é mais necessário.

Utilização

Configurar e executar a seqüência escutar/notificar usando o psql:

```
LISTEN virtual;
NOTIFY virtual;

Asynchronous NOTIFY ’virtual’ from backend with pid ’8448’ received.
```

Compatibilidade

SQL92

Não existe o comando `LISTEN` no SQL92.

LOAD

Name

`LOAD` — carrega ou recarrega um arquivo de biblioteca compartilhada

Synopsis

```
LOAD 'nome_arquivo'
```

Descrição

Carrega um arquivo de biblioteca compartilhada no espaço de endereçamento do servidor PostgreSQL. Se o arquivo tiver sido carregado anteriormente, o mesmo é descarregado primeiramente. Este comando é útil para descarregar e recarregar um arquivo de biblioteca compartilhada que foi mudado após ter sido carregado pelo servidor. Para usar a biblioteca compartilhada suas funções devem ser declaradas pelo comando *CREATE FUNCTION*.

O nome do arquivo é especificado da mesma forma que os nomes das bibliotecas compartilhadas no comando *CREATE FUNCTION*; em particular, pode-se confiar no caminho de procura e na adição automática das extensões de nomes de arquivos das bibliotecas compartilhadas do sistema padrão. Consulte o *Guia do Programador* para obter mais detalhes.

Compatibilidade

O comando `LOAD` é uma extensão do PostgreSQL.

Consulte também

CREATE FUNCTION, *Guia do Programador do PostgreSQL*

LOCK

Name

`LOCK` — bloqueia explicitamente uma tabela

Synopsis

```
LOCK [ TABLE ] nome [, ...]
LOCK [ TABLE ] nome [, ...] IN modo_de_bloqueio MODE

onde modo_de_bloqueio é um entre:

 ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE |
 SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

Entradas

nome

O nome de uma tabela existente a ser bloqueada.

ACCESS SHARE MODE

> **Note:** Este modo de bloqueio é obtido automaticamente nas tabelas sendo consultadas.

Este é o modo de bloqueio menos restritivo. Somente conflita com o modo ACCESS EXCLUSIVE. É utilizado para proteger a tabela de ser modificada por um comando `ALTER TABLE`, `DROP TABLE` ou `VACUUM FULL` concorrente.

ROW SHARE MODE

> **Note:** Automaticamente obtido pelo comando `SELECT ... FOR UPDATE`.

Conflita com os modos de bloqueio EXCLUSIVE e ACCESS EXCLUSIVE.

ROW EXCLUSIVE MODE

> **Note:** Automaticamente obtido pelos comandos `UPDATE`, `DELETE`, e `INSERT`.

Conflita com os modos de bloqueio SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE.

SHARE UPDATE EXCLUSIVE MODE

> **Note:** Automaticamente obtido pelo comando `VACUUM` (sem a opção `FULL`).

Conflita com os modos de bloqueio SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE. Este modo de bloqueio protege a tabela contra alterações concorrentes do esquema e VACUUM.

SHARE MODE

> **Note:** Automaticamente obtido pelo comando `CREATE INDEX`. Bloqueia o compartilhamento de toda a tabela.

Conflita com os modos de bloqueio ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE. Este modo de bloqueio protege a tabela contra atualização concorrente dos dados.

SHARE ROW EXCLUSIVE MODE

> **Note:** Semelhante ao EXCLUSIVE MODE, mas permite o bloqueio ROW SHARE por outros.

Conflita com os modos de bloqueio ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE.

EXCLUSIVE MODE

> **Note:** Este modo é ainda mais restritivo do que SHARE ROW EXCLUSIVE. Bloqueia todas as consultas ROW SHARE/SELECT...FOR UPDATE concorrentes.

Conflita com os modos de bloqueio ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE. Este modo de bloqueio permite somente o ACCESS SHARE concorrente, ou seja, somente leituras na tabela podem ocorrer em paralelo a uma transação contendo este tipo de bloqueio.

ACCESS EXCLUSIVE MODE

> **Note:** Automaticamente obtido pelos comandos `ALTER TABLE`, `DROP TABLE` e `VACUUM FULL`. Este é o modo de bloqueio mais restritivo, o qual protege uma tabela bloqueada de qualquer operação concorrente.

> **Note:** Este modo de bloqueio também é obtido por um comando `LOCK TABLE` não qualificado (ou seja, o comando sem a opção do modo de bloqueio explicitado).

Conflita com todos os outros modos de bloqueio.

Saídas

`LOCK TABLE`

Mensagem retornada se o bloqueio for obtido com sucesso.

`ERROR` *`nome`*`: Table does not exist.`

Mensagem retornada se *`nome`* não existir.

Descrição

O comando `LOCK TABLE` controla o acesso concorrente à tabela durante o tempo de execução da transação. O PostgreSQL sempre utiliza o modo de bloqueio menos restritivo quando é possível. O comando `LOCK TABLE` é usado nos casos onde se necessita de um modo de bloqueio mais restritivo.

Os bloqueios dos SGBDR utilizam a seguinte terminologia:

EXCLUSIVE

Um bloqueio exclusivo impede a concessão de outros bloqueios do mesmo tipo (Nota: O modo ROW EXCLUSIVE não segue esta convenção de nome perfeitamente, porque é compartilhado no nível de tabela; é exclusivo apenas com relação às linhas específicas que estão sendo atualizadas).

SHARE

Um bloqueio compartilhado permite que outros também obtenham o mesmo tipo de bloqueio, mas impede que o bloqueio EXCLUSIVE correspondente seja concedido.

ACCESS

Bloqueia o esquema da tabela.

ROW

Bloqueia linhas individualmente.

Por exemplo, suponha que um aplicativo processe uma transação no nível de isolamento READ COMMITTED e precise garantir a persistência dos dados da tabela durante a transação. Para conseguir esta persistência, pode-se obter o modo de bloqueio SHARE na tabela antes de realizar a consulta. Este procedimento impede alterações de dados concorrentes e garante que as operações de leitura posteriores na tabela acessam os dados no estado atual, porque o modo de bloqueio SHARE conflita com qualquer modo de bloqueio ROW EXCLUSIVE necessário para escrever. O comando `LOCK TABLE` *`nome`* `IN SHARE`

`MODE` aguarda até que todas as operações de escrita concorrentes terminem efetivando ou desfazendo suas transações. Portanto, quando este bloqueio é conseguido, não existe nenhuma operação de escrita sendo executada.

> **Note:** Para ler os dados em seu estado corrente atual ao executar uma transação no nível de isolamento SERIALIZABLE, é necessário executar o comando LOCK TABLE antes de executar qualquer comando da DML. A visão dos dados de uma transação serializável é congelada no momento em que o primeiro comando da DML começa a executar.

Além dos requisitos acima, se uma transação altera os dados da tabela, então o modo de bloqueio SHARE ROW EXCLUSIVE deve ser obtido para prevenir a ocorrência da condição de “impasse” (deadlock), quando duas transações concorrentes bloqueiam a tabela em modo SHARE e depois tentam mudar os dados desta tabela, as duas (implicitamente) solicitando o modo de bloqueio ROW EXCLUSIVE que conflita com o modo de bloqueio SHARE concorrente.

Para que não ocorra a situação de “impasse” (quando duas transações ficam aguardando uma pela outra) descrita acima, devem ser seguidas duas regras gerais que evitam as condições de impasse:

- As transações têm que obter o bloqueio dos mesmos objetos na mesma ordem.

  Por exemplo, se um aplicativo atualiza a linha R1 e depois atualiza a linha R2 (na mesma transação), então um segundo aplicativo não deve atualizar a linha R2 se for atualizar a linha R1 depois (na mesma transação). Em vez disto, o segundo aplicativo deve atualizar as linha R1 e R2 na mesma ordem do primeiro aplicativo.

- As transações devem obter dois modos de bloqueio conflitantes somente se um deles é auto-conflitante (ou seja, pode ser obtido por somente uma única transação de cada vez). Se diversos modos de bloqueio estão envolvidos, então as transações devem sempre solicitar o modo mais restritivo primeiro.

  Um exemplo desta regra foi dado anteriormente ao se discutir o uso do modo SHARE ROW EXCLUSIVE em vez do modo SHARE.

> **Note:** O PostgreSQL detecta “impasses” (deadlocks) e desfaz pelo menos uma das transações em espera para resolver o impasse.

Ao se bloquear várias tabelas, o comando LOCK a, b; é equivalente a LOCK a; LOCK b;. As tabelas são bloqueadas uma por uma na ordem especificada pelo comando `LOCK`.

Notas

O comando `LOCK ... IN ACCESS SHARE MODE` requer o privilégio `SELECT` na tabela especificada. Todas as outras formas de `LOCK` requerem os privilégios `UPDATE` e/ou `DELETE`.

O comando `LOCK` é útil apenas dentro de um bloco de transação (`BEGIN`...`COMMIT`), porque o bloqueio é liberado logo que a transação termina. Um comando `LOCK` aparecendo fora de um bloco de transação forma uma transação auto-contida, portanto o bloqueio será liberado tão logo seja obtido.

Utilização

Este exemplo mostra o bloqueio no modo SHARE da tabela que contém a chave primária, antes de fazer uma inserção na tabela que contém a chave estrangeira:

```
BEGIN WORK;
LOCK TABLE filmes IN SHARE MODE;
SELECT id FROM filmes
    WHERE nome = 'Star Wars: Episode I - The Phantom Menace';
-- Fazer o ROLLBACK se a linha não for encontrada
INSERT INTO filmes_comentarios VALUES
    (_id_, 'Maravilhoso! Eu estava aguardando por isto há muito tempo!');
COMMIT WORK;
```

Obter o bloqueio no modo SHARE ROW EXCLUSIVE da tabela que contém a chave primária antes de realizar a operação de exclusão:

```
BEGIN WORK;
LOCK TABLE filmes IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM filmes_comentarios WHERE id IN
    (SELECT id FROM filmes WHERE avaliacao < 5);
DELETE FROM filmes WHERE avaliacao < 5;
COMMIT WORK;
```

Compatibilidade

SQL92

Não existe o comando `LOCK TABLE` no SQL92, que em seu lugar usa o comando `SET TRANSACTION` para especificar os níveis de concorrência das transações. O PostgreSQL também suporta esta funcionalidade; Consulte o comando *SET TRANSACTION* para obter mais detalhes.

Exceto pelos modos de bloqueio ACCESS SHARE, ACCESS EXCLUSIVE e SHARE UPDATE EXCLUSIVE, os modos de bloqueio do PostgreSQL e a sintaxe do comando `LOCK TABLE` são compatíveis com as presentes no Oracle(TM).

MOVE

Name

`MOVE` — posiciona o cursor em uma determinada linha da tabela

Synopsis

```
MOVE [ direção ] [ contador ]
    { IN | FROM } cursor
```

Descrição

O comando `MOVE` permite ao usuário mover a posição do cursor o número especificado de linhas. O comando `MOVE` trabalha como o comando `FETCH`, porém somente posiciona o cursor sem retornar as linhas.

Consulte o comando *FETCH* para obter detalhes sobre a sintaxe e a utilização.

Notas

O comando `MOVE` é uma extensão da linguagem do PostgreSQL.

Consulte o comando *FETCH* para obter uma descrição dos argumentos válidos. Consulte o comando *DECLARE* para definir o cursor. Consulte os comandos *BEGIN*, *COMMIT*, e *ROLLBACK* para obter mais informações sobre as transações.

Utilização

Criar e usar um cursor:

```
    BEGIN WORK;
    DECLARE liahona CURSOR  FOR SELECT * FROM filmes;
    -- Saltar as primeiras 5 linhas:
    MOVE FORWARD 5 IN liahona;
    MOVE
    -- Ler a sexta linha no cursor liahona:
    FETCH 1 IN liahona;
    FETCH

 cod   | titulo | did | data_prod | tipo | duracao
-------+--------+-----+-----------+------+---------
 P_303 | 48 Hrs | 103 | 1982-10-22| Ação |  01:37
(1 row)
    -- fechar o cursor liahona e efetivar a transação:
    CLOSE liahona;
    COMMIT WORK;
```

Compatibilidade

SQL92

Não existe o comando `MOVE` no SQL92. Em vez disto, o SQL92 permite usar o comando `FETCH` para buscar uma linha na posição absoluta do cursor, movendo implicitamente o cursor para a posição correta.

NOTIFY

Name

`NOTIFY` — gera uma notificação

Synopsis

```
NOTIFY nome
```

Entradas

nome_da_notificação

Condição de notificação a ser sinalizada.

Saídas

`NOTIFY`

Constata que o comando NOTIFY foi executado.

Notify events

Eventos são enviados para os clientes escutando; se, e como, cada aplicativo cliente reage depende da sua programação.

Descrição

O comando `NOTIFY` envia um evento de notificação para cada aplicativo cliente que tenha executado anteriormente o comando `LISTEN` *nome_da_notificação* para a condição de notificação especificada, no banco de dados corrente.

A notificação passada para o cliente por um evento de notificação inclui o nome da condição de notificação e o PID do processo servidor que está notificando. Fica por conta do projetista do banco de dados definir os nomes das condições que serão utilizadas em um determinado banco de dados e o significado de cada uma.

Usualmente, o nome da condição de notificação é o mesmo nome de uma tabela do banco de dados, e o evento de notificação essencialmente diz “Eu modifiquei a tabela, dê uma olhada nela e veja o que há de novo”. Porém este tipo de associação não é exigida pelos comandos `NOTIFY` e `LISTEN`. Por exemplo, o

projetista do banco de dados pode usar vários nomes diferentes de condição para sinalizar diferentes tipos de mudança em uma única tabela.

O comando `NOTIFY` fornece uma forma de sinal simples, ou mecanismo de IPC (comunicação entre processos), para um conjunto de processos acessando o mesmo banco de dados do PostgreSQL. Mecanismos de nível mais alto podem ser construídos usando-se tabelas no banco de dados para passar informações adicionais (além do mero nome da condição) do notificador para o(s) ouvinte(s).

Quando o `NOTIFY` é utilizado para sinalizar a ocorrência de mudanças em uma tabela em particular, uma técnica de programação útil é colocar o `NOTIFY` em uma regra que é disparada pela atualização da tabela. Deste modo, a notificação acontece automaticamente quando a tabela é modificada, e o programador do aplicativo não vai poder acidentalmente esquecer de fazê-la.

O `NOTIFY` interage com as transações SQL de forma importante. Em primeiro lugar, se um `NOTIFY` for executado de dentro de uma transação, o evento de notificação não é enviado até que, e a menos que, a transação seja efetivada. Isto é apropriado porque, se a transação for abortada, deseja-se que nenhum de seus comandos tenha surtido efeito, incluindo o `NOTIFY`. Mas isto pode ser desconcertante se for esperado que os eventos de notificação sejam enviados imediatamente. Em segundo lugar, se um processo servidor na escuta recebe um sinal de notificação enquanto está executando uma transação, o evento de notificação não será enviado ao cliente conectado antes da transação ser completada (seja efetivada ou abortada). Novamente o raciocínio é que se a notificação for enviada de dentro de uma transação que for abortada posteriormente, vai se desejar que a notificação seja desfeita de alguma maneira --- mas o processo servidor não pode “pegar de volta” uma notificação após enviá-la para o cliente. Portanto, os eventos de notificação somente são enviados entre transações. O resultado final disto é que os aplicativos que utilizam o `NOTIFY` para sinalização de tempo real devem tentar manter as suas transações curtas.

O `NOTIFY` se comporta como os sinais do Unix em um aspecto importante: se o mesmo nome de condição for sinalizado diversas vezes em uma sucessão rápida, os ouvintes podem receber somente um evento de notificação para várias execuções do `NOTIFY`. Portanto, é uma má idéia depender do número de notificações recebidas. Em vez disso, use o `NOTIFY` para acordar os aplicativos que devam prestar atenção em algo, e use um objeto do banco de dados (como uma seqüência) para registrar o que aconteceu ou quantas vezes aconteceu.

É comum para um cliente que executa o `NOTIFY` estar escutando o mesmo nome de notificação. Neste caso vai ser recebido de volta o evento de notificação da mesma maneira que todos os outros clientes na escuta. Dependendo da lógica do aplicativo, isto pode resultar em um trabalho sem utilidade --- por exemplo, a releitura da tabela do banco de dados para encontrar as mesmas atualizações que foram feitas. No PostgreSQL 6.4, e nas versões posteriores, é possível evitar este trabalho extra verificando se o PID do processo servidor que fez a notificação (fornecido na mensagem do evento de notificação) é o mesmo PID do seu processo servidor (disponível pela libpq). Quando forem idênticos, o evento de notificação é o seu próprio trabalho retornando, podendo ser ignorado (A despeito do que foi dito no parágrafo precedente, esta é uma técnica segura. O PostgreSQL mantém as auto-notificações separadas das notificações vindas de outros processos servidores, portanto não é possível perder-se uma notificação externa ignorando-se as próprias notificações).

Notas

`nome` pode ser qualquer cadeia de caracteres válida como um nome; não é necessário que corresponda ao nome de qualquer tabela existente. Se *`nome`* estiver entre aspas (`"`), não é necessário nem que seja um nome com uma sintaxe válida; pode ser qualquer cadeia de caracteres com até 31 caracteres.

Em algumas versões anteriores do PostgreSQL, o *nome* tinha que vir entre aspas quando não correspondia a nenhum nome de tabela existente, mesmo que sintaticamente fosse um nome válido, mas não é mais necessário.

Nas versões do PostgreSQL anteriores a 6.4, o PID do processo servidor que enviou a mensagem de notificação era sempre o PID do processo servidor do próprio cliente. Então não era possível distinguir entre as próprias notificações e as notificações dos outros clientes nestas versões antigas.

Utilização

Configurar e executar a seqüência escutar/notificar usando o psql:

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous NOTIFY ’virtual’ from backend with pid ’8448’ received.
```

Compatibilidade

SQL92

Não existe o comando `NOTIFY` no SQL92.

REINDEX

Name

REINDEX — reconstrói índices corrompidos

Synopsis

```
REINDEX { TABLE | DATABASE | INDEX } nome [ FORCE ]
```

Entradas

TABLE

Reconstrói todos os índices da tabela especificada.

DATABASE

Reconstrói todos os índices do sistema do banco de dados especificado (Os índices das tabelas dos usuários não são incluídos).

INDEX

Reconstrói o índice especificado.

nome

O nome da tabela, banco de dados ou índice a ser reindexado.

FORCE

Força a reconstrução dos índices do sistema. Sem esta palavra chave o comando `REINDEX` pula os índices do sistema que não estão marcados como inválidos. FORCE é irrelevante para o comando `REINDEX INDEX`, ou para reindexar índices do usuário.

Saídas

`REINDEX`

Mensagem retornada se a tabela for reindexada com sucesso.

Descrição

O comando `REINDEX` é utilizado para reconstruir índices corrompidos. Embora na teoria nunca deve haver esta necessidade, na prática os índices podem se tornar corrompidos devido a erros de programação ou falhas nos equipamentos. O comando `REINDEX` fornece um método de recuperação.

Se houver a suspeita de que um índice de uma tabela do usuário está corrompido, pode-se simplesmente reconstruir este índice, ou todos os índices da tabela, usando o comando `REINDEX INDEX` ou `REINDEX TABLE`.

> **Note:** Outra forma de tratar o problema de índice corrompido em tabela do usuário é simplesmente eliminá-lo e recriá-lo. Pode-se preferir esta forma para manter alguma aparência de uma operação normal em uma tabela. O comando `REINDEX` obtém um bloqueio exclusivo da tabela, enquanto o comando `CREATE INDEX` bloqueia a escrita mas não a leitura da tabela.

A situação se torna mais difícil quando é necessário recuperar um índice corrompido de uma tabela do sistema. Neste caso é importante que o servidor efetuando a recuperação não esteja usando nenhum dos índices suspeitos (Sem dúvida, neste tipo de cenário pode acontecer do servidor estar caindo durante a inicialização por depender do índice corrompido). Para executar a recuperação segura, o postmaster deve ser parado e um servidor autônomo (stand-alone) do PostgreSQL deve ser iniciado fornecendo-se as opções de linha de comando -O e -P (estas opções permitem modificar as tabelas do sistema e faz com que os índices do sistema não sejam utilizados, respectivamente). Então se executa o comando `REINDEX INDEX`, `REINDEX TABLE`, ou `REINDEX DATABASE` dependendo de quanto se deseja reconstruir. Na dúvida deve ser utilizado o comando `REINDEX DATABASE FORCE` para forçar a reconstrução de todos os índices do sistema no banco de dados. Em seguida o servidor autônomo deve ser parado e o postmaster reiniciado.

Como esta é provavelmente a única situação em que a maioria das pessoas usa um servidor autônomo, algumas observações devem ser feitas:

- Inicie o servidor com um comando como

  ```
  postgres -D $PGDATA -O -P meu_banco_de_dados
  ```

  Forneça o caminho correto para a área de banco de dados na opção `-D`, ou assegure-se de que a variável de ambiente `PGDATA` está declarada. Também deve ser especificado o nome do banco de dados em particular em que se deseja trabalhar.
- Pode ser executado qualquer comando SQL e não apenas o `REINDEX`.
- Tenha em mente que o servidor autônomo trata o caractere de nova-linha como término da entrada do comando; não existe a inteligência sobre o ponto-e-vírgula como existe no psql. Para um comando prosseguir através de várias linhas, deve-se digitar uma contrabarra precedendo cada caractere de nova-linha, exceto da última. Também não existe nenhuma das conveniências do processamento da linha lida (não existe o histórico dos comandos, por exemplo).
- Para sair do servidor digite EOF (geralmente control-D).

Consulte a página de referência do postgres para obter mais informações.

Utilização

Reconstruir os índices da tabela `minha_tabela`:

```
REINDEX TABLE minha_tabela;
```

Reconstruir um único índice:

```
REINDEX INDEX meu_indice;
```

Reconstruir todos os índices do sistema (somente funciona utilizando um servidor autônomo):

```
REINDEX DATABASE meu_banco_de_dados FORCE;
```

Compatibilidade

SQL92

Não existe o comando `REINDEX` no SQL92.

RESET

Name

`RESET` — atribui a um parâmetro de tempo de execução o seu valor padrão

Synopsis

```
RESET variável

RESET ALL
```

Entradas

variável

O nome de um parâmetro de tempo de execução. Consulte o comando *SET* para ver a relação.

ALL

Atribui a todos os parâmetros de tempo de execução o seu valor padrão.

Descrição

O comando `RESET` restaura os parâmetros de tempo de execução atribuindo seus valores padrão. Consulte o comando *SET* para obter detalhes. O comando `RESET` é uma forma alternativa para

```
SET variável TO DEFAULT
```

Diagnósticos

Consulte o comando *SET*.

Exemplos

Atribuir a DateStyle o seu valor padrão:

```
RESET DateStyle;
```

Atribuir a Geqo o seu valor padrão:

```
RESET GEQO;
```

Compatibilidade

O comando `RESET` é uma extensão do PostgreSQL à linguagem.

REVOKE

Name

REVOKE — revoga privilégios de acesso

Synopsis

```
REVOKE { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIGGER } [,...]
    ON [ TABLE ] objeto [, ...]
    FROM { nome_do_usuário | GROUP nome_do_grupo | PUBLIC } [, ...]
```

Descrição

O comando REVOKE permite ao criador de um objeto revogar as permissões concedidas anteriormente a um ou mais usuários ou grupos de usuários. A palavra chave PUBLIC refere-se ao grupo de todos os usuários definido implicitamente.

Observe que um usuário em particular possui a soma dos privilégios concedidos diretamente ao próprio usuário, com os privilégios concedidos aos grupos de que for membro e com os privilégios concedidos a PUBLIC. Daí, por exemplo, revogar o privilégio SELECT para PUBLIC não significa, necessariamente, que todos os usuários perdem o privilégio SELECT sobre o objeto: àqueles que receberam o privilégio diretamente, ou através de um grupo, permanecem com o privilégio.

Consulte a descrição do comando *GRANT* para conhecer o significado dos tipos de privilégio.

Notas

Use o comando \z do psql para exibir os privilégios concedidos nos objetos existentes. Consulte também o comando GRANT para obter informações sobre o formato.

Exemplos

Revogar o privilégio de inserir na tabela filmes concedido a todos os usuários:

```
REVOKE INSERT ON filmes FROM PUBLIC;
```

Revogar todos os privilégios concedidos ao usuário manuel sobre a visão vis_tipos:

```
REVOKE ALL PRIVILEGES ON vis_tipos FROM manuel;
```

Compatibilidade

SQL92

As notas sobre compatibilidade presentes no comando *GRANT* se aplicam de forma análoga ao comando `REVOKE`. O sumário da sintaxe é:

```
REVOKE [ GRANT OPTION FOR ] { SELECT | INSERT | UPDATE | DELETE | REFERENCES }
    ON objeto [ ( coluna [, ...] ) ]
    FROM { PUBLIC | nome_do_usuário [, ...] }
    { RESTRICT | CASCADE }
```

Se user1 conceder um privilégio WITH GRANT OPTION para o user2, e user2 concedê-lo para o user3, então, user1 pode revogar este privilégio em cascata utilizando a palavra chave CASCADE. Se user1 conceder um privilégio WITH GRANT OPTION para o user2, e user2 concedê-lo ao user3, então, se user1 tentar revogar este privilégio especificando a palavra chave RESTRICT o comando irá falhar.

Consulte também

GRANT

ROLLBACK

Name

ROLLBACK — aborta a transação corrente

Synopsis

```
ROLLBACK [ WORK | TRANSACTION ]
```

Entradas

Nenhuma.

Saídas

ABORT

Mensagem retornada se o comando for executado com sucesso.

NOTICE: ROLLBACK: no transaction in progress

Se não houver nenhuma transação sendo executada.

Descrição

O comando ROLLBACK desfaz a transação corrente, fazendo com que todas as modificações realizadas pela transação sejam rejeitadas.

Notas

Use o comando *COMMIT* para terminar uma transação com sucesso. O comando *ABORT* é um sinônimo para o comando ROLLBACK.

Utilização

Para abortar todas as modificações:

```
ROLLBACK WORK;
```

Compatibilidade

SQL92

O SQL92 somente especifica as duas formas `ROLLBACK` e `ROLLBACK WORK`. Fora isso é totalmente compatível.

SELECT

Name

`SELECT` — retorna linhas de uma tabela ou de uma visão

Synopsis

```
SELECT [ ALL | DISTINCT [ ON ( expressão [, ...] ) ] ]
    * | expressão [ AS nome_saída ] [, ...]
    [ FROM item_de [, ...] ]
    [ WHERE condição ]
    [ GROUP BY expressão [, ...] ]
    [ HAVING condição [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expressão [ ASC | DESC | USING operador ] [, ...] ]
    [ FOR UPDATE [ OF nome_da_tabela [, ...] ] ]
    [ LIMIT { contador | ALL } ]
    [ OFFSET início ]

onde item_de pode ser:

[ ONLY ] nome_da_tabela [ * ]
    [ [ AS ] aliás [ ( lista_coluna_alias ) ] ]
|
( select )
    [ AS ] aliás [ ( lista_coluna_alias ) ]
|
item_de [ NATURAL ] tipo_de_junção item_de
    [ ON condição_de_junção | USING ( lista_coluna_junção ) ]
```

Entradas

expressão

O nome de uma coluna da tabela ou uma expressão.

nome_saída

Especifica outro nome para uma coluna retornada utilizando a cláusula AS. Este nome é utilizado, principalmente, como o título da coluna exibida. Também pode ser utilizado para fazer referência ao valor da coluna nas cláusulas ORDER BY e GROUP BY. Mas o *nome_saída* não pode ser usado nas cláusulas WHERE e HAVING; nestes casos deve-se escrever novamente a expressão.

item_de

A referência a uma tabela, uma subconsulta ou uma cláusula de junção. Veja abaixo para obter detalhes.

`condição`

Uma expressão booleana produzindo um resultado falso ou verdadeiro. Veja a descrição das cláusulas WHERE e HAVING abaixo.

`select`

Um comando SELECT com todas as suas funcionalidades, exceto as cláusulas ORDER BY, FOR UPDATE e LIMIT (mesmo estas podem ser utilizadas quando o SELECT está entre parênteses).

Os itens da cláusula FROM podem conter:

`nome_da_tabela`

O nome de uma tabela ou de uma visão existente. Se ONLY for especificado, somente esta tabela é consultada. Se ONLY não for especificado a tabela, e todas as suas tabelas descendentes porventura existentes, serão consultadas. O * pode ser apensado ao nome da tabela para indicar que as tabelas descendentes devem ser consultadas, mas na versão corrente este é o comportamento padrão (Nas versões anteriores a 7.1 ONLY era o comportamento padrão).

`aliás`

Um nome substituto para o `nome_da_tabela` precedente. Um aliás é utilizado para abreviar ou para eliminar ambigüidade em autojunções (onde a mesma tabela é referenciada várias vezes). Se um aliás for escrito, uma lista de aliás de coluna também pode ser escrita para fornecer nomes substitutos para uma ou mais colunas da tabela.

`select`

Uma subconsulta pode aparecer na cláusula FROM, agindo como se sua saída fosse criada como uma tabela temporária pela duração do comando SELECT. Observe que a subconsulta deve estar entre parênteses, e que um aliás *deve* ser fornecido para esta subconsulta.

`tipo_de_junção`

Um entre `[ INNER ] JOIN`, `LEFT [ OUTER ] JOIN`, `RIGHT [ OUTER ] JOIN`, `FULL [ OUTER ] JOIN`, ou `CROSS JOIN`. Para os tipos de junção INNER e OUTER, exatamente um entre NATURAL, ON `condição_de_junção`, ou USING (`lista_coluna_junção`) deve estar presente. Para CROSS JOIN, nenhum destes itens pode aparecer.

`condição_de_junção`

Uma condição de qualificação, similar à condição WHERE, exceto que somente se aplica aos dois itens sendo unidos por esta cláusula JOIN.

`lista_coluna_junção`

A condição USING lista de colunas (a, b, ...) é uma forma abreviada da condição ON tabela_esquerda.a = tabela_direita.a AND tabela_esquerda.b = tabela_direita.b ...

Saídas

Linhas

O conjunto completo das linhas resultantes da especificação da consulta.

contador

A quantidade de linhas retornadas pela consulta.

Descrição

O comando `SELECT` retorna linhas de uma ou mais tabelas. As linhas que satisfazem a condição WHERE são candidatas para seleção; se WHERE for omitido, todas as linhas são candidatas (Veja *A cláusula WHERE*).

Na verdade, as linhas retornadas não são as linhas produzidas pelas cláusulas FROM/WHERE/GROUP BY/HAVING diretamente; mais precisamente, as linhas da saída são formadas computando-se as expressões de saída do SELECT para cada linha selecionada. O `*` pode ser escrito na lista de saída como uma abreviação para todas as colunas das linhas selecionadas. Pode-se escrever também *nome_da_tabela*`.*` como uma abreviação para as colunas provenientes de apenas uma tabela.

A opção `DISTINCT` elimina as linhas repetidas do resultado, enquanto que a opção `ALL` (o padrão) retorna todas as linhas candidatas, incluindo as repetidas.

`DISTINCT ON` elimina as linhas que correspondem a todas as expressões especificadas, mantendo apenas a primeira linha de cada conjunto de repetidas. As expressões do DISTINCT ON são interpretadas usando as mesmas regras dos itens do ORDER BY; veja abaixo. Observe que a “primeira linha” de cada conjunto não pode ser prevista, a menos que `ORDER BY` seja usado para garantir que a linha desejada apareça primeiro. Por exemplo,

```
SELECT DISTINCT ON (local) local, data, condicao
FROM tbl_condicao_climatica
ORDER BY local, data DESC;
```

exibe a condição climática mais recente para cada local, mas se ORDER BY não tivesse sido usado para forçar uma ordem descendente dos valores da data para cada local, teria sido obtido um relatório com datas aleatórias para cada local.

A cláusula GROUP BY permite dividir a tabela em grupos de linhas que correspondem a um ou mais valores (Veja *A cláusula GROUP BY*).

A cláusula HAVING permite selecionar somente os grupos de linhas que atendem a uma condição específica (Veja *A cláusula HAVING*).

A cláusula ORDER BY faz com que as linhas retornadas sejam classificadas na ordem especificada. Se ORDER BY não for especificado, as linhas retornam na ordem que o sistema considera mais fácil de gerar (Veja *A cláusula ORDER BY*).

As consultas SELECT podem ser combinadas usando os operadores UNION, INTERSECT e EXCEPT. Use parênteses, se for necessário, para determinar a ordem destes operadores.

O operador UNION computa a coleção das linhas retornadas pelas consultas envolvidas. As linhas duplicadas são eliminadas, a não ser que ALL seja especificado (Veja *A cláusula UNION*).

O operador INTERSECT computa as linhas que são comuns às duas consultas (interseção). As linhas duplicadas são eliminadas, a não ser que ALL seja especificado (Veja *A cláusula INTERSECT*).

O operador EXCEPT computa as linhas que são retornadas pela primeira consulta, mas que não são retornadas pela segunda consulta. As linhas duplicadas são eliminadas, a não ser que ALL seja especificado (Veja *A cláusula EXCEPT*).

A cláusula FOR UPDATE permite ao comando SELECT realizar o bloqueio exclusivo das linhas selecionadas.

A cláusula LIMIT permite que retorne para o usuário apenas um subconjunto das linhas produzidas pela consulta (Veja *A cláusula LIMIT*).

É necessário possuir o privilégio SELECT na tabela para poder ler seus valores (Consulte os comandos `GRANT` e `REVOKE`).

A cláusula FROM

A cláusula FROM especifica uma ou mais tabelas de origem para o SELECT. Se múltiplas tabelas de origem forem especificadas o resultado será, conceitualmente, o produto Cartesiano de todas as linhas de todas estas tabelas -- mas, geralmente, condições de qualificação são adicionadas para restringir as linhas retornadas a um pequeno subconjunto do produto Cartesiano.

Quando o item da cláusula FROM é simplesmente o nome de uma tabela, implicitamente são incluídas as linhas das subtabelas desta tabela (filhas que herdam). Especificando-se `ONLY` causa a supressão das linhas das subtabelas da tabela. Antes do PostgreSQL 7.1 este era o comportamento padrão, e a adição das subtabelas era feita anexando-se um `*` ao nome da tabela. Este comportamento antigo está disponível através do comando `SET SQL_Inheritance TO OFF;`

Um item da cláusula FROM pode ser também uma subconsulta entre parênteses (note que uma cláusula aliás é exigida para a subconsulta!). Esta característica é extremamente útil porque esta é a única maneira de se obter múltiplos níveis de agrupamento, agregação ou ordenação em uma única consulta.

Finalmente, um item da cláusula FROM pode ser uma cláusula JOIN, que combina dois itens do FROM (Use parênteses, se for necessário, para determinar a ordem de aninhamento).

O CROSS JOIN e o INNER JOIN são um produto Cartesiano simples, o mesmo que seria obtido listando-se os dois itens no nível superior do FROM. O CROSS JOIN é equivalente ao INNER JOIN ON (TRUE), ou seja, nenhuma linha é removida pela qualificação. Estes tipos de junção são apenas uma notação conveniente, porque não fazem nada que não poderia ser feito usando simplesmente o FROM e o WHERE.

O LEFT OUTER JOIN retorna todas as linhas do produto Cartesiano qualificado (i.e., todas as linhas combinadas que passam pela condição ON), mais uma cópia de cada linha da tabela à esquerda para a qual não há uma linha da tabela à direita que tenha passado pela condição ON. Esta linha da tabela à esquerda é estendida por toda a largura da tabela combinada inserindo-se nulos para as colunas da tabela à direita. Observe que somente as condições ON ou USING do próprio JOIN são consideradas na hora de decidir quais linhas possuem correspondência. Condições ON ou WHERE externas são aplicadas depois.

De forma inversa, o RIGHT OUTER JOIN retorna todas as linhas da junção, mais uma linha para cada linha da tabela à direita sem correspondência (estendida com nulos na tabela à esquerda). Isto é apenas uma conveniência da notação, porque poderia ser convertido em um LEFT OUTER JOIN trocando-se a tabela à direita pela tabela à esquerda.

O FULL OUTER JOIN retorna todas as linhas da junção, mais uma linha para cada linha da tabela à esquerda sem correspondência (estendida com nulos na tabela à direita), mais uma linha da tabela à direita sem correspondência (estendida com nulos na tabela à esquerda)

Para todos os tipos de JOIN, exceto CROSS JOIN, deve-se escrever exatamente um entre ON `condição_de_junção`, USING (`lista_coluna_junção`), ou NATURAL. A cláusula ON é o caso mais geral: pode ser escrita qualquer expressão de qualificação envolvendo as duas tabelas da junção. A forma USING lista de colunas (a, b, ...) é uma abreviação para a condição ON tabela_esquerda.a = tabela_direita.a AND tabela_esquerda.b = tabela_direita.b ... Além disso, USING implica em que somente uma coluna de cada par de colunas equivalentes será incluída na saída do JOIN, e não as duas. NATURAL é uma abreviação para USING quando a lista menciona todas as colunas das tabelas com mesmo nome.

A cláusula WHERE

A condição opcional WHERE possui a forma geral:

```
WHERE expressão_booleana
```

A `expressão_booleana` pode ser qualquer expressão que retorna um valor booleano. Em muitos casos esta expressão possui a forma:

```
expressão op_condição expressão
```

ou

```
op_logico expressão
```

onde `op_condição` pode ser um entre: =, <, <=, >, >= ou <>, um operador condicional como ALL, ANY, IN, LIKE, ou um operador definido localmente, e `op_logico` pode ser um entre: AND, OR e NOT. O SELECT ignora todas as linhas para as quais a condição WHERE não retorna TRUE.

A cláusula GROUP BY

O GROUP BY especifica uma tabela agrupada derivada da aplicação desta cláusula:

```
GROUP BY expressão [, ...]
```

O GROUP BY condensa em uma única linha todas as linhas selecionadas que compartilham os mesmos valores para as colunas agrupadas. As funções de agregação, caso existam, são computadas através de todas as linhas que pertencem a cada grupo, produzindo um valor separado para cada grupo (enquanto que

sem GROUP BY, uma função de agregação produz um único valor computado através de todas as linhas selecionadas). Quando GROUP BY está presente, não é válido uma expressão de saída do SELECT fazer referência a uma coluna não agrupada, exceto dentro de uma função de agregação, porque pode haver mais de um valor possível retornado para uma coluna não agrupada.

Um item do GROUP BY pode ser o nome de uma coluna da entrada, o nome ou o número ordinal de uma coluna da saída (expressão SELECT), ou pode ser uma expressão arbitrária formada pelos valores das colunas da entrada. No caso de haver ambigüidade, o nome no GROUP BY vai ser interpretado como o sendo o nome de uma coluna da entrada, e não como o nome de uma coluna da saída.

A cláusula HAVING

A condição opcional HAVING possui a forma geral:

```
HAVING expressão_booleana
```

onde *expresão_booleana* é a mesma que foi especificada para a cláusula WHERE.

HAVING especifica uma tabela agrupada derivada pela eliminação das linhas agrupadas que não satisfazem a *expressão_booleana*. HAVING é diferente de WHERE: WHERE filtra individualmente as linhas antes da aplicação do GROUP BY, enquanto HAVING filtra os grupos de linhas criados pelo GROUP BY.

Cada coluna referenciada na *expressão_booleana* deve referenciar, sem ambigüidade, uma coluna de agrupamento, a menos que a referência apareça dentro de uma função de agregação.

A cláusula ORDER BY

```
ORDER BY expressão [ ASC | DESC | USING operador ] [, ...]
```

Um item do ORDER BY pode ser o nome ou o número ordinal de uma coluna da saída (expressão SELECT), ou pode ser uma expressão arbitrária formada por valores da coluna de entrada. No caso de haver ambigüidade, o nome no ORDER BY será interpretado como o nome de uma coluna da saída.

O número ordinal refere-se à posição ordinal (da esquerda para a direita) da coluna do resultado. Esta característica torna possível definir uma ordenação baseada em uma coluna que não possui um nome próprio. Nunca isto é absolutamente necessário, porque é sempre possível atribuir um nome a uma coluna do resultado usando a cláusula AS, como no exemplo abaixo:

```
SELECT titulo, data_prod + 1 AS newlen FROM filmes ORDER BY newlen;
```

Também é possível efetuar um ORDER BY por expressões arbitrárias (uma extensão ao SQL92), incluindo campos que não aparecem na lista de resultados do SELECT. Portanto, o seguinte comando é válido:

```
SELECT nome FROM distribuidores ORDER BY cod;
```

Uma limitação desta funcionalidade é que uma cláusula ORDER BY aplicada ao resultado de uma consulta com UNION, INTERSECT, ou EXCEPT pode apenas especificar um nome ou um número de coluna, mas não uma expressão.

Observe que se um item do ORDER BY for um nome simples que corresponda tanto ao nome de uma coluna do resultado quanto ao nome de uma coluna da entrada, o ORDER BY vai interpretar como sendo o nome da coluna do resultado. Esta opção é oposta a que é feita pelo GROUP BY na mesma situação. Esta inconsistência é determinada pelo padrão SQL92.

Opcionalmente pode ser utilizada a palavra chave DESC (descendente) ou ASC (ascendente) após cada nome de coluna na cláusula ORDER BY. Se nada for especificado, ASC é assumido por padrão. Alternativamente, um nome de operador de ordenação específico pode ser especificado. ASC é equivalente a USING $<$ e DESC é equivalente a USING $>$.

O valor nulo possui posição de ordenação maior do que qualquer outro valor do domínio. Em outras palavras, na ordenação ascendente os valores nulos ficam no final, e na ordenação descendente os valores nulos ficam no início.

A cláusula UNION

```
tabela_consulta UNION [ ALL ] tabela_consulta
    [ ORDER BY expressão [ ASC | DESC | USING operador ] [, ...] ]
    [ LIMIT { contador | ALL } ]
    [ OFFSET início ]
```

onde *tabela_consulta* especifica qualquer expressão de seleção sem as cláusulas ORDER BY, FOR UPDATE, ou LIMIT (ORDER BY e LIMIT podem aparecer na subexpressão se estiver entre parênteses. Sem os parênteses estas cláusulas se aplicam ao resultado da UNION, e não na sua expressão de entrada da direita).

O operador UNION computa a coleção (conjunto união) das linhas retornadas pelas consultas envolvidas. Os dois SELECT que representam os operandos diretos do UNION devem produzir o mesmo número de colunas, e as colunas correspondentes devem possuir tipos de dado compatíveis.

O resultado de uma UNION não contém nenhuma linha repetida, a menos que a opção ALL all seja especificada. ALL impede a eliminação das linhas repetidas.

Quando existem vários operadores UNION no mesmo comando SELECT, estes são processados da esquerda para a direita, a menos que outra ordem seja definida pelo uso de parênteses.

Atualmente, FOR UPDATE não deve ser especificado nem no resultado nem nas entradas de uma UNION.

A cláusula INTERSECT

```
tabela_consulta INTERSECT [ ALL ] tabela_consulta
    [ ORDER BY expressão [ ASC | DESC | USING operador ] [, ...] ]
    [ LIMIT { contador | ALL } ]
```

```
        [ OFFSET início ]
```

onde *tabela_consulta* especifica qualquer expressão de seleção sem as cláusulas ORDER BY, FOR UPDATE, ou LIMIT.

INTERSECT é semelhante a UNION, exceto que retorna somente as linhas presentes nas saídas das duas consultas, em vez de todas as linhas presentes nas duas consultas.

O resultado do INTERSECT não possui nenhuma linha repetida, a menos que a opção ALL seja especificada. Com ALL, uma linha que possui m repetições em L e n repetições em R aparece min(m,n) vezes.

Quando existem vários operadores INTERSECT no mesmo comando SELECT, estes são processados da esquerda para a direita, a menos que os parênteses estabeleçam outra ordem. INTERSECT tem prevalência sobre UNION --- ou seja, A UNION B INTERSECT C é processado como A UNION (B INTERSECT C), a menos que outra ordem seja estabelecida pelo uso de parênteses..

A cláusula EXCEPT

```
tabela_consulta EXCEPT [ ALL ] tabela_consulta
    [ ORDER BY expressão [ ASC | DESC | USING operador ] [, ...] ]
    [ LIMIT { contador | ALL } ]
    [ OFFSET início ]
```

onde *tabela_consulta* especifica qualquer expressão de seleção sem as cláusulas ORDER BY, FOR UPDATE, ou LIMIT.

EXCEPT é semelhante a UNION, exceto que retorna somente as linhas presentes na saída da consulta à esquerda, mas que não estão presentes na saída da consulta à direita.

O resultado do EXCEPT não possui nenhuma linha repetida, a menos que a opção ALL seja especificada. Com ALL, uma linha que possui m repetições em L e n repetições em R aparece max(m-n,0) vezes.

Quando existem vários operadores EXCEPT no mesmo comando SELECT, estes são processados da esquerda para a direita, a menos que os parênteses estabeleçam outra ordem. EXCEPT possui o mesmo nível de precedência de UNION.

A cláusula LIMIT

```
        LIMIT { contador | ALL }
        OFFSET início
```

onde *contador* especifica o número máximo de linhas retornadas, e *início* especifica o número de linhas a serem saltadas antes de começar a retornar as linhas.

O contador LIMIT permite que seja retornada apenas uma parte das linhas que são geradas pelo resultado da consulta. Se um contador limite for fornecido, não será retornado mais do que este número de linhas. Se um deslocamento for especificado, este número de linhas será saltado antes de começar o retorno das linhas.

Quando LIMIT é usado, aconselha-se utilizar a cláusula ORDER BY para colocar as linhas do resultado dentro de uma ordem única. De outra maneira, será obtido um subconjunto das linhas da consulta impossível de ser previsto --- pode-se estar querendo obter da décima a vigésima linha, mas da décima a vigésima linha de qual ordenação? Não é possível saber qual será a ordenação, a menos que ORDER BY seja especificado.

A partir do PostgreSQL 7.0, o otimizador de consultas leva LIMIT em consideração ao gerar o plano para a consulta, então é muito provável que sejam obtidos planos diferentes (resultando em ordenações diferentes das linhas) dependendo do que seja fornecido para LIMIT e OFFSET. Portanto, utilizar valores diferentes para LIMIT/OFFSET para selecionar subconjuntos diferentes do resultado de uma consulta *vai produzir resultados inconsistentes*, a menos que seja exigida uma ordenação previsível dos resultados utilizando ORDER BY. Isto não é um erro (bug), isto é uma conseqüência do fato de que o SQL não retorna os resultados de uma consulta em nenhuma ordem específica, a não ser que ORDER BY seja utilizado para definir esta ordem.

Utilização

Para efetuar a junção da tabela `filmes` com a tabela `distribuidores`:

```
SELECT f.titulo, f.did, d.nome, f.data_prod, f.tipo
    FROM distribuidores d, filmes f
    WHERE f.did = d.did

           titulo          | did |   nome           | data_prod  | tipo
---------------------------+-----+------------------+------------+----------
 The Third Man             | 101 | British Lion     | 1949-12-23 | Drama
 The African Queen         | 101 | British Lion     | 1951-08-11 | Romance
 Une Femme est une Femme   | 102 | Jean Luc Godard  | 1961-03-12 | Romance
 Vertigo                   | 103 | Paramount        | 1958-11-14 | Ação
 Becket                    | 103 | Paramount        | 1964-02-03 | Drama
 48 Hrs                    | 103 | Paramount        | 1982-10-22 | Ação
 War and Peace             | 104 | Mosfilm          | 1967-02-12 | Drama
 West Side Story           | 105 | United Artists   | 1961-01-03 | Musical
 Bananas                   | 105 | United Artists   | 1971-07-13 | Comédia
 Yojimbo                   | 106 | Toho             | 1961-06-16 | Drama
 There’s a Girl in my Soup | 107 | Columbia         | 1970-06-11 | Comédia
 Taxi Driver               | 107 | Columbia         | 1975-05-15 | Ação
 Absence of Malice         | 107 | Columbia         | 1981-11-15 | Ação
 Storia di una donna       | 108 | Westward         | 1970-08-15 | Romance
 The King and I            | 109 | 20th Century Fox | 1956-08-11 | Musical
 Das Boot                  | 110 | Bavaria Atelier  | 1981-11-11 | Drama
 Bed Knobs and Broomsticks | 111 | Walt Disney      |            | Musical
(17 rows)
```

Obter soma da coluna `duracao` de todos os filmes, agrupando os resultados por `tipo`:

```
SELECT tipo, SUM(duracao) AS total FROM filmes GROUP BY tipo;

   tipo   | total
```

```
----------+-------
 Ação     | 07:34
 Comédia  | 02:58
 Drama    | 14:28
 Musical  | 06:42
 Romance  | 04:38
(5 rows)
```

Obter a soma da coluna `duracao` de todos os filmes, agrupando os resultados por `tipo`, mostrando apenas os grupos com total inferior a 5 horas:

```
SELECT tipo, SUM(duracao) AS total
    FROM filmes
    GROUP BY tipo
    HAVING SUM(duracao) < INTERVAL '5 hour';

 tipo     | total
----------+-------
 Comédia  | 02:58
 Romance  | 04:38
(2 rows)
```

Os dois exemplos mostrados abaixo são formas idênticas de ordenação dos resultados de acordo com o conteúdo da segunda coluna (`nome`):

```
SELECT * FROM distribuidores ORDER BY nome;
SELECT * FROM distribuidores ORDER BY 2;

 did |       nome
-----+------------------
 109 | 20th Century Fox
 110 | Bavaria Atelier
 101 | British Lion
 107 | Columbia
 102 | Jean Luc Godard
 113 | Luso filmes
 104 | Mosfilm
 103 | Paramount
 106 | Toho
 105 | United Artists
 111 | Walt Disney
 112 | Warner Bros.
 108 | Westward
(13 rows)
```

Este exemplo mostra como obter a união das tabelas `distribuidores` e `atores`, restringindo o resultado aos nomes que iniciam pela letra W em cada uma das tabelas. Somente linhas distintas são desejadas, por isso a palavra chave ALL é omitida:

```
distribuidores:               atores:
 did |     nome              id |     nome
-----+--------------        ----+----------------
 108 | Westward               1 | Woody Allen
 111 | Walt Disney            2 | Warren Beatty
 112 | Warner Bros.           3 | Walter Matthau
 ...                         ...

SELECT distribuidores.nome
    FROM   distribuidores
    WHERE  distribuidores.nome LIKE 'W%'
UNION
SELECT atores.nome
    FROM   atores
    WHERE  atores.nome LIKE 'W%';

      nome
----------------
 Walt Disney
 Walter Matthau
 Warner Bros.
 Warren Beatty
 Westward
 Woody Allen
```

Compatibilidade

Extensões

O PostgreSQL permite que seja omitida a cláusula `FROM` da consulta. Esta funcionalidade da linguagem de consulta PostQuel original foi mantida, sendo muita utilizada para calcular os resultados de expressões com constantes:

```
SELECT 2+2;

 ?column?
----------
        4
```

Outros SGBDRs não podem fazer isto, a não ser que seja introduzida uma tabela especial com uma única linha para se efetuar a seleção a partir desta tabela. Uma utilização menos óbvia desta característica é para abreviar consultas normais a uma ou mais tabelas:

```
SELECT distribuidores.* WHERE distribuidores.nome = 'Westward';

 did | nome
-----+----------
 108 | Westward
```

Isso funciona porque um FROM implícito é adicionado para cada tabela que é referenciada na consulta mas não é mencionada no FROM. Ao mesmo tempo em que é uma forma conveniente de abreviar, é fácil de ser usado de forma errada. Por exemplo, a consulta

```
SELECT distribuidores.* FROM distribuidores d;
```

deve ser um engano; provavelmente o que se deseja é

```
SELECT d.* FROM distribuidores d;
```

em vez da junção sem restrições

```
SELECT distribuidores.* FROM distribuidores d, distribuidores distribuidores;
```

que é obtido na realidade. Para ajudar a detectar este tipo de engano, o PostgreSQL 7.1, e posteriores, advertem quando um FROM implícito é usado em uma consulta que também possui uma cláusula FROM explícita.

SQL92

Cláusula SELECT

No padrão SQL92, a palavra chave opcional AS é somente informativa podendo ser omitida sem afetar o significado. O analisador (parser) do PostgreSQL requer a presença desta palavra chave para mudar o nome das colunas da saída, porque a funcionalidade de extensibilidade de tipo ocasiona ambigüidades no analisador dentro deste contexto. Entretanto, AS é opcional nos itens da cláusula FROM.

A frase DISTINCT ON não faz parte do SQL92, assim como LIMIT e OFFSET.

No SQL92, uma cláusula ORDER BY pode utilizar somente os nomes das colunas do resultado, ou números, enquanto uma cláusula GROUP BY pode utilizar somente os nomes das colunas da entrada. O PostgreSQL estende cada uma destas cláusulas para permitir a outra escolha também (mas utiliza a interpretação padrão no caso de ambigüidade). O PostgreSQL também permite que as duas cláusulas especifiquem expressões arbitrárias. Observe que os nomes que aparecem em uma expressão são sempre obtidos a partir dos nomes das colunas da entrada, e não dos nomes das colunas do resultado.

Cláusulas UNION/INTERSECT/EXCEPT

A sintaxe do SQL92 para UNION/INTERSECT/EXCEPT permite a adição da opção CORRESPONDING BY:

```
tabela_consulta UNION [ALL]
    [CORRESPONDING [BY (nome_coluna [,...])]]
    tabela_consulta
```

A cláusula CORRESPONDING BY não é suportada pelo PostgreSQL.

SELECT INTO

Name

`SELECT INTO` — cria uma nova tabela a partir do resultado de uma consulta

Synopsis

```
SELECT [ ALL | DISTINCT [ ON ( expressão [, ...] ) ] ]
    * | expressão [ AS nome_saída ] [, ...]
    INTO [ TEMPORARY | TEMP ] [ TABLE ] nova_tabela
    [ FROM item_de [, ...] ]
    [ WHERE condição ]
    [ GROUP BY expressão [, ...] ]
    [ HAVING condição [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expressão [ ASC | DESC | USING operador ] [, ...] ]
    [ FOR UPDATE [ OF nome_tabela [, ...] ] ]
    [ LIMIT [ início , ] { contador | ALL } ]
    [ OFFSET início ]

onde item_de pode ser:

[ ONLY ] nome_tabela [ * ]
    [ [ AS ] alias [ ( lista_alias_coluna ) ] ]
|
( select )
    [ AS ] alias [ ( lista_alias_coluna ) ]
|
item_de [ NATURAL ] join_type item_de
    [ ON condição_junção | USING ( lista_coluna_junção ) ]
```

Entradas

TEMPORARY
TEMP

Se TEMPORARY ou TEMP for especificado, a tabela criada vai existir apenas durante a sessão, sendo automaticamente eliminada no fim da sessão. Uma tabela permanente com o mesmo nome, caso exista, não será visível (nesta sessão) enquanto a tabela temporária existir. Todo índice criado em tabela temporária também é temporário.

nova_tabela

O nome da nova tabela a ser criada. A tabela não pode existir. Entretanto, pode ser criada uma tabela temporária que possua o mesmo nome de uma tabela permanente existente.

Todas as outras entradas estão descritas detalhadamente no comando SELECT.

Saídas

Consulte os comandos CREATE TABLE e SELECT para obter um sumário das mensagens de saída possíveis.

Descrição

O comando `SELECT INTO` cria uma nova tabela e a preenche com os dados produzidos por uma consulta. Os dados não são retornados para o cliente, como acontece em um comando `SELECT` normal. As colunas da nova tabela possuem os mesmos nomes e tipos de dado das colunas de saída do comando `SELECT`.

> **Note:** O comando CREATE TABLE AS é funcionalmente equivalente ao comando `SELECT INTO`. `CREATE TABLE AS` é a sintaxe recomendada, porque `SELECT INTO` não é padrão. De fato, esta forma do `SELECT INTO` não está disponível no PL/pgSQL e no ecpg , porque os dois interpretam a cláusula INTO de forma diferente.

Compatibilidade

O SQL92 utiliza o comando `SELECT ... INTO` para representar a seleção de valores para dentro de variáveis escalares do programa hospedeiro, em vez de criar uma nova tabela. Esta é a mesma utilização encontrada no PL/pgSQL e no ecpg . A utilização no PostgreSQL do comando `SELECT INTO` para representar a criação de uma tabela é histórica. É melhor utilizar o comando `CREATE TABLE AS` para esta finalidade nos programas novos (O comando `CREATE TABLE AS` também não é padrão, mas tem menos chance de gerar confusão).

SET

Name

SET — muda um parâmetro de tempo de execução

Synopsis

```
SET variável { TO | = } { valor | 'valor' | DEFAULT }
SET TIME ZONE { 'zona_horária' | LOCAL | DEFAULT }
```

Entradas

variável

Um parâmetro de tempo de execução cujo valor pode ser mudado.

valor

O novo valor do parâmetro. Pode ser usado `DEFAULT` para especificar a atribuição do valor padrão do parâmetro. São permitidas listas de cadeias de caracteres, mas construções mais complexas podem precisar estar entre apóstrofos (`'`) ou entre aspas (`"`).

Descrição

O comando `SET` altera os parâmetros de configuração de tempo de execução. Os seguintes parâmetros podem ser alterados:

CLIENT_ENCODING
NAMES

Define a codificação multibyte do cliente. A codificação especificada deve ser suportada pelo servidor.

Esta opção somente está disponível quando o PostgreSQL for gerado com suporte para multibyte.

DATESTYLE

Escolhe o estilo de representação da data/hora. Duas atribuições separadas são feitas: o padrão de saída para data/hora, e a interpretação da entrada ambígua.

Estes são os estilos de saída para data/hora:

ISO

Usa o estilo ISO 8601 para data e hora (`YYYY-MM-DD HH:MM:SS`). Este é o padrão.

SQL

Usa o estilo Oracle/Ingres para data e hora. Observe que este estilo não tem relação com o SQL (que define o estilo ISO 8601). O nome desta opção é apenas um acidente histórico.

PostgreSQL

Usa o formato tradicional do PostgreSQL.

German

Usa `dd.mm.yyyy` para representar as datas numéricas.

As duas opções a seguir determinam o subestilo dos formatos de saída “SQL” e “PostgreSQL”, e a interpretação preferencial da entrada de data ambígua.

European

Usa `dd/mm/yyyy` para representar as datas numéricas.

NonEuropean
US

Usa `mm/dd/yyyy` para representar as datas numéricas.

O valor para `SET DATESTYLE` pode ser um da primeira relação (estilos de saída), ou um da segunda relação (subestilos), ou um de cada separados por vírgula.

A inicialização do formato da data pode ser feita por:

Atribuição da variável de ambiente `PGDATESTYLE`. Se PGDATESTYLE estiver definida no ambiente do cliente baseado na `libpq`, automaticamente a `libpq` atribui a DATESTYLE o valor de PGDATESTYLE durante a inicialização da conexão.
Executar o postmaster utilizando a opção `-o -e` para atribuir às datas a convenção `European`.

A opção `DateStyle` é prevista apenas para portar aplicativos. Para formatar os valores de data/hora conforme a necessidade, deve-se usar a família de funções `to_char`.

SEED

Define a semente interna para o gerador de números randômicos.

`valor`

O valor da semente a ser usada pela função `random`. Os valores permitidos são números de ponto flutuante entre 0 e 1, os quais são então multiplicados por 2^{31}-1. Este produto irá estourar a capacidade sem dar mensagem de erro se um número fora deste intervalo for usado.

A semente também pode ser definida através da função `setseed` do SQL:

```
SELECT setseed(valor);
```

SERVER_ENCODING

Define a codificação multibyte do servidor.

Esta opção somente está disponível quando o PostgreSQL for gerado com suporte para multibyte.

TIME ZONE
TIMEZONE

Define a zona horária padrão para a sessão. Os argumentos podem ser uma constante SQL de intervalo de tempo, uma constante inteira ou de precisão dupla, ou uma cadeia de caracteres representando uma zona horária suportada pelo sistema operacional hospedeiro.

Os valores possíveis para a zona horária dependem do sistema operacional. Por exemplo, no Linux o arquivo `/usr/share/zoneinfo` contém um banco de dados de zona horária.

Abaixo estão mostrados alguns valores válidos para a zona horária:

’PST8PDT’

Zona horária para a Califórnia.

’Portugal’

Zona horária para Portugal.

’Europe/Rome’

Zona horária para a Itália.

7

Zona horária com 7 horas de deslocamento à oeste do GMT (equivalente à PDT).

INTERVAL ’08:00’ HOUR TO MINUTE

Zona horária com 8 horas de deslocamento à oeste do GMT (equivalente à PST).

LOCAL
DEFAULT

Define como sendo a zona horária local (a zona horária do sistema operacional).

Se uma zona horária inválida for especificada, a zona horária torna-se GMT (na maioria dos sistemas).

Se a variável de ambiente `PGTZ` tiver valor atribuído numa estação do cliente baseada na `libpq`, automaticamente a `libpq` atribui à TIMEZONE o valor de `PGTZ` durante a inicialização da conexão.

Uma relação mais extensa dos parâmetros de tempo de execução pode ser encontrada no *Guia do Administrador*.

Use o comando *SHOW* para ver os valores correntes estabelecidos para os parâmetros.

Diagnósticos

`SET VARIABLE`

Mensagem retornada se o comando for executado com sucesso.

`ERROR: not a valid option name:` *`nome`*

O parâmetro que se tentou definir não existe.

`ERROR: permission denied`

É necessário ser um superusuário para ter acesso a certas definições.

`ERROR:` *`nome`* `can only be set at start-up`

Certos parâmetros não podem ser mudados após o servidor ter sido inicializado.

Exemplos

Definir o estilo de data tradicional do PostgreSQL com convenções européias:

```
SET DATESTYLE TO PostgreSQL,European;
```

Definir a zona horária de Berkeley, Califórnia, usando aspas para preservar os atributos maiúsculos do especificador de zona horária (note que aqui o formato da data/hora é ISO):

```
SET TIME ZONE "PST8PDT";
SELECT CURRENT_TIMESTAMP AS hoje;

         hoje
------------------------
 1998-03-31 07:41:21-08
```

Definir a zona horária para a Itália (observe que é necessário o uso de apóstrofos ou de aspas para tratar caracteres especiais):

```
SET TIME ZONE ’Europe/Rome’;
SELECT CURRENT_TIMESTAMP AS hoje;

         hoje
------------------------
 1998-03-31 17:41:31+02
```

Compatibilidade

SQL92

A segunda sintaxe mostrada acima (`SET TIME ZONE`) tenta imitar o SQL92. Entretanto, o SQL permite somente deslocamentos numéricos para a zona horária. Todos os outros parâmetros mostrados, assim como a primeira sintaxe mostrada acima, são extensões do PostgreSQL.

SET CONSTRAINTS

Name

`SET CONSTRAINTS` — especifica o modo de restrição da transação corrente

Synopsis

```
SET CONSTRAINTS { ALL | restrição [, ...] } { DEFERRED | IMMEDIATE }
```

Descrição

O comando `SET CONSTRAINTS` especifica o comportamento da avaliação da restrição na transação corrente. No modo `IMMEDIATE` (imediato), as restrições são verificadas ao final de cada comando. No modo `DEFERRED` (postergado), as restrições não são verificadas até a efetivação (`commit`) da transação.

Na hora da criação, é sempre dada à restrição uma destas três características: `INITIALLY DEFERRED` (inicialmente postergada), `INITIALLY IMMEDIATE DEFERRABLE` (inicialmente imediata, postergável), ou `INITIALLY IMMEDIATE NOT DEFERRABLE` (inicialmente imediata, não postergável). A terceira classe não é afetada pelo comando `SET CONSTRAINTS`.

Atualmente, somente as restrições de chave estrangeira são afetadas por este comando. As restrições de verificação (`check`) e de unicidade são sempre inicialmente imediata não postergável.

Compatibilidade

SQL92, SQL99

`SET CONSTRAINT` é definida no SQL92 e no SQL99.

SET SESSION AUTHORIZATION

Name

`SET SESSION AUTHORIZATION` — define o identificador do usuário da sessão e o identificador do usuário corrente, da sessão corrente.

Synopsis

```
SET SESSION AUTHORIZATION 'nome_do_usuário'
```

Descrição

Este comando define o identificador do usuário da sessão e o identificador do usuário corrente, do contexto da sessão SQL corrente, como sendo *`nome_do_usuário`*.

O identificador do usuário da sessão é inicialmente definido como sendo o (possivelmente autenticado) nome do usuário fornecido pelo cliente. O identificador do usuário corrente normalmente é igual ao identificador do usuário da sessão, mas pode mudar temporariamente no contexto das funções “setuid” e de outros mecanismos semelhantes. O identificador do usuário corrente é relevante para a verificação das permissões.

A execução deste comando é permitida apenas se o usuário inicial da sessão (o *usuário autenticado*) tiver o privilégio de superusuário. Esta permissão é mantida durante o período da conexão; por exemplo, é possível se tornar temporariamente um usuário sem privilégios e, em seguida, voltar a ser um superusuário.

Exemplos

```
SELECT SESSION_USER, CURRENT_USER;
 current_user | session_user
--------------+--------------
 pedro        | pedro

SET SESSION AUTHORIZATION 'paulo';

SELECT SESSION_USER, CURRENT_USER;
 current_user | session_user
--------------+--------------
 paulo        | paulo
```

Compatibilidade

SQL99

O SQL99 permite que algumas outras expressões apareçam no lugar de *`nome_do_usuário`*, as quais não são importantes na prática. O PostgreSQL permite a sintaxe do identificador (`"nome_do_usuário"`),

que o SQL não permite. O SQL não permite a execução deste comando durante uma transação; O PostgreSQL não faz esta restrição porque não há motivo para fazê-la. O padrão deixa os privilégios necessários para executar este comando por conta da implementação.

SET TRANSACTION

Name

`SET TRANSACTION` — define as características da transação corrente

Synopsis

```
SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL
    { READ COMMITTED | SERIALIZABLE }
```

Descrição

Este comando define o nível de isolamento da transação. O comando `SET TRANSACTION` define as características para a transação SQL corrente, não possuindo qualquer efeito sobre nenhuma transação seguinte. Este comando não pode ser utilizado após a primeira consulta ou instrução que modifique os dados (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `FETCH`, `COPY`) da transação tiver sido executada. O comando `SET SESSION CHARACTERISTICS` define o nível de isolamento padrão para todas as transações da sessão. O comando `SET TRANSACTION` pode substitui-lo para uma transação individual.

O nível de isolamento de uma transação determina quais dados a transação pode enxergar quando outras transações estão processando ao mesmo tempo.

READ COMMITTED

Um comando enxerga apenas as linhas efetivadas (`commit`) antes do início da sua execução. Este é o padrão.

SERIALIZABLE

A transação corrente enxerga apenas as linhas efetivadas (`commit`) antes da primeira consulta ou instrução que modifique os dados ter sido executada nesta transação.

> **Tip:** Intuitivamente serializável significa que, duas transações concorrentes deixam o banco de dados no mesmo estado que estas duas transações, executadas uma após a outra em qualquer ordem, deixaria.

Notas

O nível de isolamento padrão da transação também pode ser definido pelo comando

```
SET default_transaction_isolation = 'valor'
```

no arquivo de configuração. Consulte o *Guia do Administrador* para obter mais informações.

Compatibilidade

SQL92, SQL99

SERIALIZABLE é o nível de isolamento padrão do SQL. O PostgreSQL não possui os níveis de isolamento `READ UNCOMMITTED` e `REPEATABLE READ`. Devido ao controle de concorrência multi-versão, o nível serializável não é verdadeiramente serializável. Consulte o *Guia do Usuário* para obter mais informações.

No SQL existem outras duas características da transação que podem ser definidas com este comando: se a transação é de leitura apenas e o tamanho da área de diagnósticos. Nenhum destes conceitos é suportado pelo PostgreSQL.

SHOW

Name

SHOW — mostra o valor de um parâmetro de tempo de execução

Synopsis

```
SHOW nome

SHOW ALL
```

Entradas

nome

O nome de um parâmetro de tempo de execução. Consulte o comando *SET* para ver a relação.

ALL

Mostra todos os parâmetros da sessão corrente.

Descrição

O comando SHOW mostra o valor corrente de um parâmetro de tempo de execução. Estas variáveis podem ser definidas pelo comando SET, ou são determinadas na inicialização do servidor.

Diagnósticos

`ERROR: not a valid option name: nome`

Mensagem retornada quando a *nome* não corresponde a um parâmetro existente.

`ERROR: permission denied`

É necessário ser um superusuário para poder ver certos parâmetros.

`NOTICE: Time zone is unknown`

Se as variáveis de ambiente TZ ou PGTZ não tiverem valor atribuído.

Exemplos

Mostrar o valor do `DateStyle` (estilo da data) corrente:

```
SHOW DateStyle;
NOTICE:  DateStyle is ISO with US (NonEuropean) conventions
```

Mostrar o estado do otimizador genético corrente (`geqo`):

```
SHOW GEQO;
NOTICE:  geqo is on
```

Compatibilidade

O comando `SHOW` é uma extensão do PostgreSQL à linguagem.

TRUNCATE

Name

`TRUNCATE` — esvazia a tabela

Synopsis

```
TRUNCATE [ TABLE ] nome
```

Entradas

nome

O nome da tabela a ser truncada.

Saídas

`TRUNCATE`

Mensagem retornada se a tabela for truncada com sucesso.

Descrição

O comando `TRUNCATE` remove rapidamente todas as linhas da tabela. Tem o mesmo efeito do comando `DELETE` sem a cláusula WHERE, mas como não varre a tabela é mais rápido. É mais vantajoso para tabelas grandes.

O comando `TRUNCATE` não pode ser utilizado dentro de um bloco de transação (delimitado por `BEGIN`/`COMMIT`), porque não existe a possibilidade de desfazê-lo.

Utilização

Truncar a tabela `tbl_grande`:

```
TRUNCATE TABLE tbl_grande;
```

Compatibilidade

SQL92

Não existe o comando `TRUNCATE` no SQL92.

UNLISTEN

Name

UNLISTEN — pára de escutar uma notificação

Synopsis

```
UNLISTEN { nome_notificação | * }
```

Entradas

nome_notificação

O nome de uma condição de notificação registrada previamente.

*

Todos os registros de escuta atuais deste processo servidor são removidos.

Saídas

UNLISTEN

Constata que o comando foi executado.

Descrição

O comando UNLISTEN é utilizado para remover um registro de NOTIFY existente. UNLISTEN cancela qualquer registro existente da sessão corrente do PostgreSQL para escutar a condição de notificação *nome_notificação*. A condição especial * (curinga) cancela todos os registros de escuta da sessão corrente.

O comando *NOTIFY* contém uma discussão mais extensa da utilização do comando LISTEN e do comando NOTIFY.

Notas

O *nome_notificação* não necessita ser um nome de classe válido, podendo ser qualquer cadeia de caracteres válida como um nome, com até 32 caracteres.

O servidor não reclama quando é executado o `UNLISTEN` para algo que não esteja sendo escutado. Cada processo servidor executa automaticamente o comando `UNLISTEN *` ao encerrar sua execução.

Utilização

Para participar de um registro existente:

```
LISTEN virtual;
LISTEN
NOTIFY virtual;
NOTIFY
Asynchronous NOTIFY ’virtual’ from backend with pid ’8448’ received
```

Quando UNLISTEN é executado, os comandos NOTIFY posteriores são ignorados:

```
UNLISTEN virtual;
UNLISTEN
NOTIFY virtual;
NOTIFY
-- notice no NOTIFY event is received
```

Compatibilidade

SQL92

Não existe o comando `UNLISTEN` no SQL92.

UPDATE

Name

UPDATE — atualiza linhas de uma tabela

Synopsis

```
UPDATE [ ONLY ] tabela SET coluna = expressão [, ...]
    [ FROM lista_de ]
    [ WHERE condição ]
```

Entradas

tabela

O nome de uma tabela existente.

coluna

O nome de uma coluna da *tabela*.

expressão

Uma expressão válida ou um valor a ser atribuído à coluna.

lista_de

Uma extensão não padrão do PostgreSQL que permite colunas de outras tabelas aparecerem na condição WHERE.

condição

Consulte o comando SELECT para obter uma descrição mais detalhada da cláusula WHERE.

Saídas

UPDATE #

Mensagem retornada se o comando for executado com sucesso. O # representa o número de linhas atualizadas. Se # for 0 então nenhuma linha foi atualizada.

Descrição

O comando `UPDATE` muda os valores das colunas especificadas em todas as linhas que satisfazem a condição. Somente as colunas a serem modificadas devem aparecer na relação de colunas do comando.

Referências a `arrays` utilizam a mesma sintaxe encontrada no comando *SELECT*. Assim sendo, um único elemento de um `array`, uma faixa de elementos de um `array`, ou todo o `array` pode ser substituído em um único comando.

É necessário possuir acesso de escrita na tabela para poder modificá-la, assim como acesso de leitura em todas as tabelas mencionadas na condição da cláusula WHERE.

Por padrão, o comando UPDATE atualiza todas as tuplas da tabela especificada e de suas filhas. Para atualizar apenas a tabela referenciada deve ser utilizada a cláusula ONLY.

Utilização

Mudar a palavra `Drama` por `Suspense` na coluna `tipo`:

```
UPDATE filmes
SET tipo = ’Suspense’
WHERE tipo = ’Drama’;
SELECT *
FROM filmes
WHERE tipo = ’Drama’ OR tipo = ’Suspense’;

 cod   |     titulo    | did | data_prod  |   tipo   | tempo
-------+---------------+-----+------------+----------+-------
 BL101 | The Third Man | 101 | 1949-12-23 | Suspense | 01:44
 P_302 | Becket        | 103 | 1964-02-03 | Suspense | 02:28
 M_401 | War and Peace | 104 | 1967-02-12 | Suspense | 05:57
 T_601 | Yojimbo       | 106 | 1961-06-16 | Suspense | 01:50
 DA101 | Das Boot      | 110 | 1981-11-11 | Suspense | 02:29
```

Compatibilidade

SQL92

O SQL92 define uma sintaxe distinta para o comando UPDATE na posição do cursor:

```
UPDATE tabela SET coluna = expressão [, ...]
    WHERE CURRENT OF cursor
```

onde *`cursor`* identifica um cursor aberto.

VACUUM

Name

`VACUUM` — limpa e opcionalmente analisa o banco de dados

Synopsis

```
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ tabela ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ tabela [ (coluna [, ...] ) ] ]
```

Entradas

FULL

Seleciona uma limpeza “completa”, que pode recuperar mais espaço, mas é muito mais demorada e bloqueia a tabela em modo exclusivo.

FREEZE

Seleciona um “congelamento” agressivo das tuplas.

VERBOSE

Produz um relatório detalhado da atividade de limpeza de cada tabela.

ANALYZE

Atualiza as estatísticas utilizadas pelo otimizador para determinar o modo mais eficiente de executar uma consulta.

tabela

O nome da tabela específica a ser limpa. Por padrão todas as tabelas do banco de dados corrente.

coluna

O nome da coluna específica a ser analisada. Por padrão todas as colunas.

Saídas

`VACUUM`

O comando terminou.

`NOTICE: --Relation` *tabela*`--`

O cabeçalho do relatório da *tabela*.

```
NOTICE: Pages 98: Changed 25, Reapped 74, Empty 0, New 0; Tup 1000: Vac 3000,
Crash 0, UnUsed 0, MinLen 188, MaxLen 188; Re-using: Free/Avail. Space
586952/586952; EndEmpty/Avail. Pages 0/74. Elapsed 0/0 sec.
```

A análise da *tabela*.

```
NOTICE: Index índice: Pages 28; Tuples 1000: Deleted 3000. Elapsed 0/0 sec.
```

A análise de um índice da tabela.

Descrição

O comando `VACUUM` recupera a área de armazenamento ocupada pelas tuplas excluídas. Na operação normal do PostgreSQL as tuplas que são excluídas, ou que se tornam obsoletas devido a uma atualização, não são fisicamente removidas da tabela; elas permanecem presentes até que o comando `VACUUM` seja executado. Portanto, é necessário executar o `VACUUM` periodicamente, especialmente em tabelas freqüentemente atualizadas.

Sem nenhum parâmetro, o `VACUUM` processa todas as tabelas do banco de dados corrente. Com um parâmetro, o `VACUUM` processa somente esta tabela.

O comando `VACUUM ANALYZE` executa o comando `VACUUM` e depois o comando `ANALYZE` para cada tabela selecionada. Esta é uma forma de combinação adequada para os scripts das rotinas de manutenção. Consulte o comando *ANALYZE* para obter mais detalhes sobre o seu processamento.

Somente o `VACUUM` (sem o `FULL`) simplesmente recupera o espaço e torna-o disponível para ser reutilizado. Esta forma do comando pode operar em paralelo com a leitura e escrita normal da tabela. O `VACUUM FULL` executa um processamento mais extenso, incluindo a movimentação das tuplas através de blocos para tentar compactar a tabela para o menor número de blocos de disco. Esta forma é muito mais lenta e requer o bloqueio exclusivo de cada tabela para processá-la.

O `FREEZE` é uma opção com finalidade especial, que faz as tuplas serem marcadas como “congeladas” logo que seja possível, em vez de aguardar até que sejam bastante velhas. Se for realizado quando não existir nenhuma outra transação sendo executada no mesmo banco de dados, então é garantido que todas as tuplas do banco de dados sejam “congeladas” e não estarão sujeitas aos problemas do recomeço do ID de transação, não importa quanto tempo o banco de dados for deixado sem executar o `VACUUM`. O `FREEZE` não é recomendado para uso rotineiro. Sua única utilização esperada é em conjunto com a preparação dos bancos de dados de gabarito dos usuários, ou outros bancos de dados que são unicamente para leitura e não receberão as operações da rotina de manutenção `VACUUM`. Consulte o *Guia do Administrador* para obter detalhes.

Notas

Recomenda-se que os bancos de dados de produção ativos sejam `VACUUM`-nizados freqüentemente (pelo menos toda noite), para que sejam removidas as linhas expiradas. Após incluir ou excluir um grande número de linhas, pode ser uma boa idéia executar o comando `VACUUM ANALYZE` para a tabela afetada. Esta operação vai atualizar os catálogos do sistema com os resultados de todas as mudanças recentes,

permitindo ao otimizador de consultas do PostgreSQL fazer melhores escolhas ao planejar as consultas dos usuários.

A opção `FULL` não é recomendada para uso rotineiro, mas pode ser útil em casos especiais. Um exemplo é após ter sido removida a maioria das linhas da tabela e deseja-se que a tabela seja fisicamente encolhida para ocupar menos espaço em disco. O comando `VACUUM FULL` geralmente encolhe mais a tabela do que o comando `VACUUM` simples.

Utilização

Abaixo está mostrado um exemplo da execução do comando `VACUUM` em uma tabela do banco de dados `regression`:

```
regression=> VACUUM VERBOSE ANALYZE onek;
NOTICE:  --Relation onek--
NOTICE:  Index onek_unique1: Pages 14; Tuples 1000: Deleted 3000.
        CPU 0.00s/0.11u sec elapsed 0.12 sec.
NOTICE:  Index onek_unique2: Pages 16; Tuples 1000: Deleted 3000.
        CPU 0.00s/0.10u sec elapsed 0.10 sec.
NOTICE:  Index onek_hundred: Pages 13; Tuples 1000: Deleted 3000.
        CPU 0.00s/0.10u sec elapsed 0.10 sec.
NOTICE:  Index onek_stringu1: Pages 31; Tuples 1000: Deleted 3000.
        CPU 0.01s/0.09u sec elapsed 0.10 sec.
NOTICE:  Removed 3000 tuples in 70 pages.
        CPU 0.02s/0.04u sec elapsed 0.07 sec.
NOTICE:  Pages 94: Changed 0, Empty 0; Tup 1000: Vac 3000, Keep 0, UnUsed 0.
        Total CPU 0.05s/0.45u sec elapsed 0.59 sec.
NOTICE:  Analyzing onek
VACUUM
```

Compatibilidade

SQL92

Não existe o comando `VACUUM` no SQL92.

II. Aplicativos para a estação cliente do PostgreSQL

Esta parte contém informações de referência para os aplicativos e utilitários usados na estação cliente do PostgreSQL. Nem todos os comandos são de uso geral, alguns requerem privilégios especiais para serem executados. A característica comum destes aplicativos é que podem ser executados a partir de qualquer computador, independentemente de onde o servidor de banco de dados esteja instalado.

createdb

Name

`createdb` — cria um novo banco de dados do PostgreSQL

Synopsis

`createdb` [*opções*...] [*nome_bd*] [*descrição*]

Entradas

-h, --host *hospedeiro*

Especifica o nome da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/), é considerado como sendo o diretório do soquete do domínio Unix.

-p, --port *porta*

Especifica a porta Internet TCP/IP, ou o soquete do domínio local Unix, onde o servidor está aguardando as conexões.

-U, --username *nome_do_usuário*

Nome do usuário para se conectar.

-W, --password

Força a solicitação da senha.

-e, --echo

Exibe os comandos que o createdb gera e envia para o servidor.

-q, --quiet

Não exibe a resposta.

-D, --location *diretório_de_dados*

Especifica o local alternativo de banco de dados. Consulte também o aplicativo initlocation.

-T, --template *gabarito*

Especifica o banco de dados de gabarito, a partir do qual este banco de dados será gerado.

-E, --encoding *codificação*

Especifica o esquema de codificação de caracteres a ser usado neste banco de dados.

nome_bd

Especifica o nome do banco de dados a ser criado. O nome deve ser único entre todos os bancos de dados do PostgreSQL desta instalação. O padrão é criar o banco de dados com o mesmo nome do usuário atual do sistema operacional.

`descrição`

Especifica, opcionalmente, um comentário a ser associado com o banco de dados criado.

As opções `-h`, `-p`, `-U`, `-W` e `-e` são passadas literalmente para o psql. As opções `-D`, `-T` e `-E` são convertidas em opções do comando SQL *CREATE DATABASE* subjacente; consulte este comando para obter mais informações sobre estas opções.

Saídas

`CREATE DATABASE`

O banco de dados foi criado com sucesso.

`createdb: Database creation failed.`

A criação do banco de dados falhou.

`createdb: Comment creation failed. (Database was created.)`

O comentário/descrição para o banco de dados não pôde ser criado, mas o banco de dados foi criado. Pode ser usado agora o comando SQL `COMMENT ON DATABASE` para criar o comentário.

Se houver uma condição de erro, a mensagem de erro do servidor será exibida. Consulte o comando *CREATE DATABASE* e o aplicativo psql para ver as causas possíveis.

Descrição

O createdb cria um banco de dados novo do PostgreSQL. O usuário que executa este comando se torna o dono do banco de dados.

O createdb é um script envoltório que usa o comando SQL *CREATE DATABASE* através do terminal interativo do PostgreSQL psql. Portanto, não existe nada em especial sobre criar bancos de dados desta ou daquela maneira, significando que o psql deve ser encontrado pelo script, e que o servidor de banco de dados deve estar executando na máquina de destino. Também se aplicam os padrões definidos e as variáveis de ambiente disponíveis para o psql e para a biblioteca cliente libpq.

Utilização

Para criar o banco de dados `demo` usando o servidor de banco de dados padrão:

```
$ createdb demo
CREATE DATABASE
```

A resposta é a mesma que teria sido recebida se fosse executado o comando SQL `CREATE DATABASE`.

Para criar o banco de dados `demo` usando o servidor na máquina `eden`, porta 5000, usando o esquema de codificação `LATIN1` e vendo o comando subjacente:

```
$ createdb -p 5000 -h eden -E LATIN1 -e demo
CREATE DATABASE "demo" WITH ENCODING = 'LATIN1'
CREATE DATABASE
```

createlang

Name

`createlang` — define uma nova linguagem procedural do PostgreSQL

Synopsis

`createlang` [*opções_de_conexão*...] *nome_da_linguagem* [*nome_bd*]
`createlang` [*opções_de_conexão*...] --list | -l *nome_bd*

Entradas

O createlang aceita os seguintes argumentos de linha de comando:

nome_da_linguagem

Especifica o nome da linguagem de programação procedural a ser definida.

-d, --dbname *nome_bd*

Especifica em qual banco de dados a linguagem deve ser adicionada. O padrão é usar o banco de dados com o mesmo nome do usuário atual do sistema operacional.

-e, --echo

Exibe os comandos SQL à medida que são executados.

-l, --list

Exibe a relação das linguagens instaladas no banco de dados de destino (que deve ser especificado).

--L *diretório*

Especifica o diretório onde o interpretador da linguagem deve ser encontrado. Normalmente o diretório é encontrado automaticamente; esta opção tem por finalidade a depuração.

O createlang também aceita os seguintes argumentos de linha de comando para os parâmetros de conexão:

-h, --host *hospedeiro*

Especifica o nome da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/), é considerado como sendo o diretório do soquete do domínio Unix.

-p, --port *porta*

Especifica a porta Internet TCP/IP, ou o soquete do domínio local Unix, onde o servidor está aguardando as conexões.

-U, --username *nome_do_usuário*

Nome do usuário para se conectar.

-W, --password

Força a solicitação da senha.

Saídas

A maioria das mensagens de erro são auto-explicativas. Se não for, execute o createlang com a opção `--echo` e consulte o comando SQL respectivo para obter detalhes. Consulte também o aplicativo psql para ver outras possibilidades.

Descrição

O createlang é um utilitário para adicionar uma nova linguagem de programação a um banco de dados do PostgreSQL. O createlang pode tratar todas as linguagens fornecidas junto com a distribuição padrão do PostgreSQL, mas não as linguagens fornecidas por terceiros.

Embora as linguagens de programação do servidor possam ser adicionadas diretamente usando vários comandos SQL, recomenda-se o uso do createlang porque este realiza várias verificações e é muito mais fácil de usar. Consulte o comando *CREATE LANGUAGE* para obter mais informações.

Notas

Use o droplang para remover uma linguagem.

O createlang é um script envoltório que chama o psql várias vezes. Se as coisas estiverem dispostas de uma maneira que seja requerida uma senha para se conectar, a senha será solicitada várias vezes.

Utilização

Para instalar a linguagem `pltcl` no banco de dados `template1`:

```
$ createlang pltcl template1
```

createuser

Name

`createuser` — define uma nova conta de usuário do PostgreSQL

Synopsis

`createuser` [*opções*...] [*nome_do_usuário*]

Entradas

-h, --host *hospedeiro*

Especifica o nome da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/), é considerado como sendo o diretório do soquete do domínio Unix.

-p, --port *porta*

Especifica a porta Internet TCP/IP, ou o soquete do domínio local Unix, onde o servidor está aguardando as conexões.

-e, --echo

Exibe os comandos que o createuser gera e envia para o servidor.

-q, --quiet

Não exibe a resposta.

-d, --createdb

É permitido ao novo usuário criar bancos de dados.

-D, --no-createdb

Não é permitido ao novo usuário criar bancos de dados.

-a, --adduser

É permitido ao novo usuário criar outros usuários (Nota: na verdade isto faz do novo usuário um *superusuário*. Esta opção não tem um nome adequado).

-A, --no-adduser

Não é permitido ao novo usuário criar outros usuários (ou seja, o novo usuário é um usuário normal e não um superusuário).

-P, --pwprompt

Se for fornecido, o createuser solicita a senha do novo usuário, não sendo necessário caso não se pretenda usar autenticação por senha.

-i, --sysid *uid*

Permite escolher uma identificação do usuário diferente do padrão. Embora não seja necessário, algumas pessoas gostam.

-E, --encrypted

Criptografa a senha do usuário armazenada no banco de dados. Se não for especificado, o padrão é utilizado.

-N, --unencrypted

Não criptografa a senha do usuário armazenada no banco de dados. Se não for especificado, o padrão é utilizado.

`nome_do_usuário`

Especifica o nome do usuário do PostgreSQL a ser criado. Este nome deve ser único entre todos os usuários do PostgreSQL.

Será solicitado o nome e outras informações se não forem especificadas na linha de comando.

As opções `-h`, `-p` e `-e` são passadas literalmente para o psql. As opções `-U` e `-W` do psql também estão disponíveis, mas podem causar confusão neste contexto.

Saídas

`CREATE USER`

O usuário foi criado com sucesso.

`createuser: creation of user "nome_do_usuário" failed`

Aconteceu algum erro. O usuário não foi criado.

Se houver uma condição de erro, a mensagem de erro do servidor será exibida. Consulte o comando *CREATE USER* e o aplicativo psql para ver as causas possíveis.

Descrição

O createuser cria um novo usuário do PostgreSQL. Somente os superusuários (usuários com o `usesuper` definido na tabela `pg_shadow`) podem criar novos usuários do PostgreSQL, portanto o createuser deve ser executado por alguém que seja um superusuário do PostgreSQL.

Ser um superusuário também implica na capacidade de não ser afetado pelas verificações de permissão de acesso do banco de dados, portanto o privilégio de superusuário deve ser concedido criteriosamente.

O createuser é um script envoltório que usa o comando SQL *CREATE USER* através do terminal interativo do PostgreSQL psql. Portanto, não existe nada em especial sobre criar usuários desta ou daquela maneira, significando que o psql deve ser encontrado pelo script, e que o servidor de banco de dados deve estar executando na máquina de destino. Também se aplicam os padrões definidos e as variáveis de ambiente disponíveis para o psql e para a biblioteca cliente libpq.

Utilização

Para criar o usuário `joel` no servidor de banco de dados padrão:

```
$ createuser joel
Is the new user allowed to create databases? (y/n) n
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER
```

Criar o mesmo usuário `joel` usando o servidor na máquina `eden`, porta 5000, evitando o pedido de informações e vendo o comando subjacente:

```
$ createuser -p 5000 -h eden -D -A -e joel
CREATE USER "joel" NOCREATEDB NOCREATEUSER
CREATE USER
```

dropdb

Name

`dropdb` — remove um banco de dados do PostgreSQL

Synopsis

`dropdb` [*`opções`*...] *`nome_bd`*

Entradas

-h, --host *`hospedeiro`*

Especifica o nome da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/), é considerado como sendo o diretório do soquete do domínio Unix.

-p, --port *`porta`*

Especifica a porta Internet TCP/IP, ou o soquete do domínio local Unix, onde o servidor está aguardando as conexões.

-U, --username *`nome_do_usuário`*

Nome do usuário para se conectar.

-W, --password

Força a solicitação da senha.

-e, --echo

Exibe os comandos que o dropdb gera e envia para o servidor.

-q, --quiet

Não exibe a resposta.

-i, --interactive

Solicita a confirmação antes de fazer qualquer operação destrutiva.

`nome_bd`

Especifica o nome do banco de dados a ser removido. O banco de dados deve ser um dos bancos de dados existentes no PostgreSQL desta instalação.

As opções `-h`, `-p`, `-U`, `-W` e `-e` são passadas literalmente para o psql.

Saídas

`DROP DATABASE`

O banco de dados foi removido com sucesso.

`dropdb: Database removal failed.`

Algum erro aconteceu.

Havendo uma condição de erro, a mensagem de erro do servidor é exibida. Consulte o comando *DROP DATABASE* e o aplicativo psql para ver as causas possíveis.

Descrição

O dropdb remove do PostgreSQL um banco de dados existente. Para executar este comando é necessário ser um superusuário, ou o dono do banco de dados.

O dropdb é um script envoltório que usa o comando SQL *DROP DATABASE* através do terminal interativo do PostgreSQL psql. Portanto, não existe nada em especial sobre remover bancos de dados desta ou daquela maneira, significando que o psql deve ser encontrado pelo script, e que o servidor de banco de dados deve estar executando na máquina de destino. Também se aplicam os padrões definidos e as variáveis de ambiente disponíveis para o psql e para a biblioteca cliente libpq.

Utilização

Para remover o banco de dados `demo` do servidor de banco de dados padrão:

```
$ dropdb demo
DROP DATABASE
```

Para remover o banco de dados `demo` usando o servidor na máquina `eden`, porta 5000, com confirmação e vendo o comando utilizado:

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE "demo"
DROP DATABASE
```

droplang

Name

`droplang` — remove uma linguagem procedural do PostgreSQL

Synopsis

`droplang` [*opções_de_conexão*...] *nome_da_linguagem* [*nome_do_bd*]
`droplang` [*opções_de_conexão*...] --list | -l *nome_do_bd*

Entradas

O droplang aceita os seguintes argumentos de linha de comando:

nome_da_linguagem

Especifica o nome da linguagem de programação do servidor a ser removida.

[-d, --dbname] *nome_do_bd*

Especifica de qual banco de dados a linguagem deve ser removida. O padrão é usar o banco de dados com o mesmo nome do usuário atual do sistema operacional.

-e, --echo

Exibe os comandos SQL à medida que são executados.

-l, --list

Exibe a relação das linguagens instaladas no banco de dados de destino (que deve ser especificado).

O droplang também aceita os seguintes argumentos de linha de comando para os parâmetros de conexão:

-h, --host *hospedeiro*

Especifica o nome da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/), é considerado como sendo o diretório do soquete do domínio Unix.

-p, --port *porta*

Especifica a porta Internet TCP/IP, ou o soquete do domínio local Unix, onde o servidor está aguardando as conexões.

-U, --username *nome_do_usuário*

Nome do usuário para se conectar.

-W, --password

Força a solicitação da senha.

Saídas

A maioria das mensagens de erro são auto-explicativas. Se não for, execute o droplang com a opção `--echo` e consulte o comando SQL respectivo para obter detalhes. Consulte também o aplicativo psql para ver outras possibilidades.

Descrição

O droplang é um utilitário para remover de um banco de dados do PostgreSQL uma linguagem de programação existente. O droplang pode remover qualquer linguagem procedural, mesmo àquelas não fornecidas na distribuição do PostgreSQL.

Embora as linguagens de programação do servidor possam ser removidas diretamente usando vários comandos SQL, recomenda-se usar o droplang porque este executa várias verificações e é muito mais fácil de usar. Consulte o comando *DROP LANGUAGE* para obter mais informações.

Notas

Use o createlang para adicionar uma linguagem.

Utilização

Para remover a linguagem `pltcl`:

```
$ droplang pltcl nome_do_bd
```

dropuser

Name

`dropuser` — remove uma conta de usuário do PostgreSQL

Synopsis

`dropuser` [*opções*...] [*nome_do_usuário*]

Entradas

-h, --host *hospedeiro*

Especifica o nome da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/), é considerado como sendo o diretório do soquete do domínio Unix.

-p, --port *porta*

Especifica a porta Internet TCP/IP, ou o soquete do domínio local Unix, onde o servidor está aguardando as conexões.

-e, --echo

Exibe os comandos que o dropuser gera e envia para o servidor.

-q, --quiet

Não exibe a resposta.

-i, --interactive

Solicita a confirmação antes de remover o usuário.

nome_do_usuário

Especifica o nome do usuário do PostgreSQL a ser removido. O nome deve existir na instalação PostgreSQL. Será solicitado o nome caso este não seja fornecido na linha de comando.

As opções `-h`, `-p` e `-e` são passadas literalmente para o psql. As opções `-U` e `-W` do psql também estão disponíveis, mas podem causar confusão neste contexto.

Saídas

`DROP USER`

O usuário foi removido com sucesso.

`dropuser: deletion of user "`*`nome_do_usuário`*`" failed`

Aconteceu algum erro. O usuário não foi removido.

Havendo uma condição de erro, a mensagem de erro do servidor é exibida. Consulte o comando *DROP USER* e o aplicativo psql para ver as causas possíveis.

Descrição

O dropuser remove um usuário do PostgreSQL *e* os bancos de dados que este usuário possui. Somente os usuários com `usesuper` definido na tabela `pg_shadow` podem remover usuários do PostgreSQL.

O dropuser é um script envoltório que usa o comando SQL *DROP USER* através do terminal interativo do PostgreSQL psql. Portanto, não existe nada em especial sobre remover usuários desta ou daquela maneira, significando que o psql deve ser encontrado pelo script, e que o servidor de banco de dados deve estar executando na máquina de destino. Também se aplicam os padrões definidos e as variáveis de ambiente disponíveis para o psql e para a biblioteca cliente libpq.

Utilização

Para remover o usuário `joel` do servidor de banco de dados padrão:

```
$ dropuser joel
DROP USER
```

Para remover o usuário `joel` usando o postmaster na máquina `eden`, porta 5000, com confirmação e vendo o comando utilizado:

```
$ dropuser -p 5000 -h eden -i -e joel
User "joel" and any owned databases will be permanently deleted.
Are you sure? (y/n) y
DROP USER "joel"
DROP USER
```

ecpg

Name

`ecpg` — pré-processador de SQL embutido para a linguagem C

Synopsis

`ecpg` [-v] [-t] [-I *caminho_de_inclusão*] [-o *arquivo_de_saída*] *arquivo*...

Entradas

O ecpg aceita os seguintes argumentos de linha de comando:

-v

Exibe informação da versão.

-t

Ativa a auto-efetivação (auto-commit) das transações. Neste modo, cada comando é automaticamente efetivado, a menos que esteja dentro de um bloco de transação explícito. No modo padrão, os comandos só são efetivados quando o `exec sql commit` é executado.

-I *caminho_de_inclusão*

Especifica um caminho de inclusão adicional. Por padrão os seguintes: `.` (o diretório atual), `/usr/local/include`, o caminho de inclusão que foi definido na hora da compilação do PostgreSQL (por padrão, `/usr/local/pgsql/include`) e `/usr/include`.

-o *arquivo_de_saída*

Especifica que o ecpg deve escrever toda a sua saída no *arquivo_de_saída*. Se esta opção não for fornecida, a saída é escrita em *nome*`.c`, assumindo-se que o arquivo de entrada se chama *nome*`.pgc`. Se o arquivo de entrada não tiver o sufixo esperado `.pgc`, então o arquivo de saída terá `.pgc` apensado ao nome do arquivo de entrada.

arquivo

Os arquivos a serem processados.

Saídas

O ecpg cria um arquivo ou escreve em `stdout`.

Valor retornado

O ecpg retorna 0 se terminar a execução com sucesso, ou um valor diferente de 0 se ocorrer algum erro.

Descrição

O ecpg é um pré-processador de SQL embutido para a linguagem C e o PostgreSQL. Permite o desenvolvimento de programas C com código SQL embutido.

Linus Tolke (<`linus@epact.se`>) foi o autor original do ecpg (até a versão 0.2). Michael Meskes (<`meskes@debian.org`>) é o atual autor e mantenedor do ecpg. Thomas Good (<`tomg@q8.nrnet.org`>) é o autor da última revisão do `man page` do ecpg, no qual este documento se baseia.

Utilização

Pré-processamento para Compilação

Um arquivo fonte com SQL embutido deve ser pré-processado antes de ser compilado:

```
ecpg [ -d ] [ -o arquivo ] arquivo.pgc
```

onde o sinalizador opcional `-d` ativa a depuração. A extensão `.pgc` é uma maneira arbitrária de caracterizar um fonte ecpg.

Pode-se desejar redirecionar a saída do pré-processador para um arquivo de `log`.

Compilação e Ligação

Assumindo-se que os binários do PostgreSQL estão em `/usr/local/pgsql`, será necessário compilar e ligar (link) o arquivo fonte pré-processado usando:

```
gcc -g -I /usr/local/pgsql/include [ -o arquivo ] arquivo.c -L /usr/local/pgsql/lib -
```

Gramática

Bibliotecas

O pré-processador adiciona duas diretivas ao código fonte:

```
#include <ecpgtype.h>
#include <ecpglib.h>
```

Declaração das Variáveis

As variáveis declaradas dentro do código fonte ecpg devem estar precedidas por:

```
EXEC SQL BEGIN DECLARE SECTION;
```

Analogamente, a seção de declaração das variáveis deve terminar por:

```
EXEC SQL END DECLARE SECTION;
```

Note: Antes da versão 2.1.0 cada variável tinha que ser declarada em uma linha separada. A partir da versão 2.1.0 múltiplas variáveis podem ser declaradas em uma única linha:

```
char  foo[16], bar[16];
```

Tratamento de Erros

A área de comunicação SQL é definida pelo:

```
EXEC SQL INCLUDE sqlca;
```

Note: A palavra `sqlca` deve ser escrita em letras minúsculas. Enquanto a convenção SQL pode ser seguida, ou seja, usar letras maiúsculas para separar o SQL embutido das declarações C, o `sqlca` (que inclui o arquivo de cabeçalho `sqlca.h`) *deve* estar em letras minúsculas, devido ao fato do prefixo EXEC SQL indicar que esta inclusão será feita pelo ecpg. O ecpg diferencia letras maiúsculas de minúsculas (`SQLCA.h` não será encontrado). O `EXEC SQL INCLUDE` pode ser utilizado para incluir outros arquivos de cabeçalho desde que a diferença entre as letras maiúsculas e minúsculas seja observada.

O comando `sqlprint` é utilizado junto com a declaração `EXEC SQL WHENEVER` para ativar o tratamento de erros ao longo do programa:

```
EXEC SQL WHENEVER sqlerror sqlprint;
```

e

```
EXEC SQL WHENEVER not found sqlprint;
```

> **Note:** Este *não* é um exemplo exaustivo da utilização da declaração `EXEC SQL WHENEVER`. Outros exemplos de utilização podem ser encontrados em manuais do SQL (por exemplo, *The LAN TIMES Guide to SQL* por Groff and Weinberg).

Conectar ao Servidor de Banco de Dados

Pode-se conectar ao banco de dados através de:

```
EXEC SQL CONNECT TO nome_bd;
```

onde o nome do banco de dados não é escrito entre apóstrofos (`'`). Antes da versão 2.1.0 o nome do banco de dados tinha que vir entre apóstrofos.

Também é possível especificar o nome do servidor e a porta no comando de conexão. A sintaxe é:

```
nome_bd[@servidor][:porta]
```

ou

```
<tcp|unix>:postgresql://servidor[:porta][/nome_bd][?opções]
```

Comandos

Em geral, os comandos SQL aceitos por outros aplicativos como o psql podem ser embutidos no código C. Abaixo estão alguns exemplos de como fazer.

Criar Tabela:

```
EXEC SQL CREATE TABLE foo (number int4, ascii char(16));
EXEC SQL CREATE UNIQUE index num1 on foo(number);
EXEC SQL COMMIT;
```

Incluir:

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

Excluir:

```
EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;
```

Seleção de uma única linha:

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

Seleção usando Cursor:

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT number, ascii FROM foo
    ORDER BY ascii;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;
```

Atualização:

```
EXEC SQL UPDATE foo
    SET ascii = ’foobar’
    WHERE number = 9999;
EXEC SQL COMMIT;
```

Notas

A definição completa da estrutura DEVE estar listada dentro da seção de declaração.

Consulte o arquivo `TODO` (a fazer) no fonte para conhecer outras funcionalidades que estão faltando.

pgaccess

Name

`pgaccess` — um aplicativo cliente do PostgreSQL com interface gráfica

Synopsis

`pgaccess` [*nome_bd*]

Opções

nome_bd

O nome do banco de dados existente a ser acessado.

Descrição

O PgAccess fornece uma interface gráfica para o PostgreSQL através da qual pode-se gerenciar e editar tabelas, definir consultas, seqüências e funções.

O PgAccess pode:

- Abrir qualquer banco de dados usando o computador, a porta, o nome do usuário e a senha especificados.
- Executar o comando *VACUUM*.
- Salvar as preferências no arquivo `~/.pgaccessrc`.

Para as tabelas, o PgAccess pode:

- Abrir várias tabelas para visualização, com o número de linhas exibidas configurável.
- Redimensionar as colunas arrastando as linhas verticais da grade.
- Quebrar o texto nas células.
- Ajustar dinamicamente a altura da célula ao editar.
- Salvar o leiaute da tabela para todas as tabelas.
- Importar/exportar para arquivos externos (SDF, CSV).
- Usar a capacidade de filtragem; definir filtros como `valor > 3.14`.
- Especificar a ordenação; entrar manualmente os campos de ordenação.
- Editar no local; dar um duplo clique no texto a ser modificado.
- Excluir registros; apontar para o registro, pressionar a tecla **Delete**.
- Incluir novos registros; salvar a nova linha com um clique no botão direito do mouse.
- Criar tabelas com um assistente.
- Renomear e excluir (DROP) tabelas.

- Obter informação sobre as tabelas, incluindo o dono, informações sobre os campos e índices.

Para as consultas, o PgAccess pode:

- Definir, editar e armazenar consultas definidas pelo usuário.
- Salvar o leiaute da visão.
- Armazenar consultas como visões.
- Executar com parâmetros opcionais entrados pelo usuário como, por exemplo,

```
select * from faturas where ano=[parameter "Ano das faturas"]
```

- Ver o resultado de qualquer consulta de seleção.
- Executar consultas de ação (inclusão, exclusão, atualização).
- Construir consultas utilizando um construtor de consultas visual com suporte a arrastar & soltar, e aliás das tabelas.

Para as seqüências, o PgAccess pode:

- Definir novas instâncias.
- Inspecionar as instâncias existentes.
- Excluir.

Para as visões, o PgAccess pode:

- Defini-las, salvando as consultas como visões.
- Vê-las, podendo filtrar e ordenar.
- Desenvolver novas visões.
- Excluir (drop) visões existentes.

Para as funções, o PgAccess pode:

- Definir.
- Inspecionar.
- Excluir.

Para os relatórios, o PgAccess pode:

- Gerar relatórios simples de uma tabela (estágio beta).
- Mudar fonte, tamanho e estilo dos campos e dos rótulos.
- Carregar e salvar relatórios no banco de dados.
- Visualizar tabelas, impressão Postscript de exemplo.

Para os formulários, o PgAccess pode:

- Abrir formulários definidos pelo usuário.
- Utilizar um módulo de desenvolvimento de formulário.
- Acessar conjunto de registros usando assistente de consultas.

Para os scripts, o PgAccess pode:

- Definir.
- Modificar.
- Chamar scripts definidos pelo usuário.

Notas

O PgAccess é escrito em Tcl/Tk. O PostgreSQL deve ter sido compilado com suporte à Tcl (Tool Command Language) para o PgAccess poder ser usado.

pg_config

Name

`pg_config` — retorna informações sobre a versão instalada do PostgreSQL

Synopsis

`pg_config` {--bindir | --includedir | --includedir-server | --libdir | --pkglibdir | --configure | --version...}

Descrição

O utilitário pg_config exibe os parâmetros de configuração da versão do PostgreSQL atualmente instalada. Sua finalidade é, por exemplo, ser usado por pacotes de software que desejem interfacear com o PostgreSQL para encontrar os arquivos de cabeçalho e bibliotecas necessários.

Opções

Para usar o pg_config deve-se fornecer uma ou mais das seguintes opções:

`--bindir`

Exibe a localização dos executáveis do usuário. Usa-se, por exemplo, para encontrar o programa psql. Normalmente, este é também o local onde o programa `pg_config` reside.

`--includedir`

Exibe a localização dos arquivos de cabeçalho C e C++ das interfaces do cliente.

`--includedir-server`

Exibe a localização dos arquivos de cabeçalho C e C++ para a programação do servidor.

`--libdir`

Exibe a localização das bibliotecas de código objeto.

`--pkglibdir`

Exibe a localização dos módulos carregáveis dinamicamente, ou onde o servidor deve procurá-los (Também podem estar instalados neste diretório outros arquivos de dados dependentes da arquitetura).

`--configure`

Exibe as opções que foram passadas para o script `configure` quando o PostgreSQL foi configurado para ser gerado. Pode ser utilizado para reproduzir uma configuração idêntica, ou para descobrir com quais opções o pacote binário foi gerado (Entretanto, observe que os pacotes binários geralmente contêm modificações específicas da distribuição).

`--version`

Exibe a versão do PostgreSQL e termina.

Se mais de uma opção (exceto `--version`) for fornecida, a informação é exibida na mesma ordem, uma por linha.

Notas

A opção `--includedir-server` é nova no PostgreSQL 7.2. Nas versões anteriores, os arquivos de inclusão do servidor estavam instalados no mesmo local dos cabeçalhos dos clientes, que podia ser consultado pelo `--includedir`. Para tratar os dois casos, deve-se tentar primeiro a nova opção e testar o status da saída, para verificar se foi executado com sucesso.

Nas versões anteriores ao PostgreSQL 7.1, antes do comando `pg_config` existir, não existia um método equivalente para encontrar as informações de configuração.

Histórico

O utilitário `pg_config` apareceu pela primeira vez no PostgreSQL 7.1.

Consulte também

Guia do Programador do PostgreSQL

pg_dump

Name

pg_dump — extrai um banco de dados do PostgreSQL para um arquivo script ou de exportação

Synopsis

pg_dump [-a | -s] [-b] [-c] [-C] [-d | -D] [-f *arquivo*] [-F *formato*] [-i] [-n | -N] [-o] [-O] [-R] [-S] [-t *tabela*] [-v] [-x] [-X *palavra_chave*] [-Z *0...9*] [-h *hospedeiro*] [-p *porta*] [-U *nome_do_usuário*] [-W] *nome_bd*

Descrição

O pg_dump é um utilitário para salvar um banco de dados do PostgreSQL em um arquivo script ou de exportação. Os arquivos script são no formato texto-puro e contêm os comandos SQL necessários para reconstruir o banco de dados no estado em que este se encontrava no momento que foi salvo. Estes scripts podem ser usados até para reconstruir o banco de dados em outras máquinas com outras arquiteturas e, com algumas modificações, até em outros SGBDR. Além do script, existem outros formatos de exportação feitos para serem usados em conjunto com o pg_restore para reconstruir o banco de dados, que permitem ao pg_restore selecionar o que deve ser restaurado, ou mesmo reordenar a restauração dos itens. Estes formatos de exportação também são projetados para serem portáveis entre arquiteturas.

O pg_dump salva as informações necessárias para regerar todos os tipos, funções, tabelas, índices, agregações e operadores definidos pelo usuário. Adicionalmente, todos os dados são salvos no formato texto para que possam ser prontamente importados, bem como tratados por ferramentas de edição.

O pg_dump é útil para exportar o conteúdo de um banco de dados a ser movido de um PostgreSQL para outro.

Quando usado com um dos formatos de exportação e combinado com o pg_restore , o pg_dump fornece um mecanismo de exportação e importação flexível. O pg_dump pode ser usado para exportar todo o banco de dados e, posteriormente, o pg_restore pode ser usado para examinar o arquivo e/ou selecionar que partes do banco de dados devem ser importadas. O formato mais flexível produzido é o "personalizado" (custom, -Fc), que permite a seleção e a reordenação de todos os itens exportados, sendo comprimido por padrão. O formato tar (-Ft) não é comprimido e não permite reordenar os dados durante a importação mas, por outro lado, é bastante flexível; além disso, pode ser tratado por outras ferramentas como o tar.

Ao executar o pg_dump deve-se examinar a saída procurando por advertências (escritas na saída de erro padrão), atento especialmente às limitações listadas abaixo.

O pg_dump cria cópias de segurança consistentes, mesmo se o banco de dados estiver sendo usado concorrentemente. O pg_dump não bloqueia os outros usuários que porventura estejam acessando o banco de dados (leitura ou gravação).

Opções

O `pg_dump` aceita os seguintes argumentos de linha de comando (As formas longas das opções estão disponíveis apenas em algumas plataformas).

`nome_bd`

Especifica o nome do banco de dados a ser exportado.

-a
--data-only

Exporta somente os dados, não o esquema (definições dos dados).

Esta opção só faz sentido para o formato texto-puro. Para os outros formatos esta opção pode ser especificada ao se chamar o `pg_restore`.

-b
--blobs

Exporta os objetos binários grandes (blobs).

-c
--clean

Gera comandos para excluir (drop) os objetos do banco de dados antes de criá-los.

Esta opção só faz sentido para o formato texto-puro. Para os outros formatos esta opção pode ser especificada ao se chamar o `pg_restore`.

-C
--create

Inicia a saída por um comando para criar o próprio banco de dados e se conectar ao banco de dados criado (Com um script assim, não importa em qual o banco de dados se está conectado antes de executar o script).

Esta opção só faz sentido para o formato texto-puro. Para os outros formatos esta opção pode ser especificada ao se chamar o `pg_restore`.

-d
--inserts

Exporta os dados como comandos `INSERT` (em vez de `COPY`), tornando a importação muito lenta, porém os arquivos exportados tornam-se mais portáveis para outros SGBDR.

-D
--column-inserts
--attribute-inserts

Exporta os dados como comandos `INSERT` com nomes explícitos das colunas (`INSERT INTO` *`tabela`* `(`*`coluna`*`, ...) VALUES ...`), tornando a importação muito lenta, mas é necessário se for desejado mudar a ordem das colunas.

-f *`arquivo`*
--file=*`arquivo`*

Envia a saída para o arquivo especificado. Se for omitido, a saída padrão é usada.

-F *formato*
--format=*formato*

Seleciona o formato da saída. O *formato* pode ser um dos seguintes:

p

Exporta um script SQL para um arquivo texto-puro (padrão)

t

Exporta um arquivo `tar` adequado para servir de entrada para o `pg_restore`. Usando este formato de exportação pode-se reordenar e/ou excluir elementos do esquema durante a restauração do banco de dados. Também é possível limitar quais dados são importados durante a restauração.

c

Exporta um arquivo personalizado apropriado para servir de entrada para o `pg_restore`. Este é o formato mais flexível porque permite a reordenação da importação dos dados, assim como dos elementos do esquema. Este formato também é comprimido por padrão.

-i
--ignore-version

Ignora a diferença de versão entre o `pg_dump` e o servidor de banco de dados. Como o `pg_dump` trabalha muito ligado aos catálogos do sistema, toda versão do `pg_dump` é feita para ser usada somente com a versão correspondente do servidor de banco de dados. Use esta opção se for necessário desconsiderar a verificação de versão (mas, se o `pg_dump` falhar, não diga que não foi avisado).

-n
--no-quotes

Suprime as aspas (`"`) em torno dos identificadores a menos que seja absolutamente necessário. Pode causar problemas na importação dos dados exportados se existirem palavras reservadas usadas pelos identificadores. Este era o comportamento padrão para o `pg_dump` antes da versão 6.4.

-N
--quotes

Inclui aspas em torno dos identificadores. Este é o padrão.

-o
--oids

Exporta os identificadores de objeto (OIDs) para todas as tabelas. Deve-se usar esta opção quando a coluna OID é referenciada de alguma maneira (por exemplo, em uma restrição de chave estrangeira). Caso contrário esta opção não deve ser usada.

-O
--no-owner

Não gera comandos para definir o mesmo dono do objeto do banco de dados original. Tipicamente, o `pg_dump` gera comandos `\connect` (específico do `psql`) para definir o dono dos elementos do esquema. Consulte também as opções `-R` e `-X use-set-session-authorization`. Observe que

a opção `-O` não impede todas as reconexões ao banco de dados, mas somente àquelas que são usadas exclusivamente para acertar o dono.

Esta opção só faz sentido para o formato texto-puro. Para os outros formatos esta opção pode ser especificada ao se chamar o `pg_restore`.

-R
--no-reconnect

Proíbe o `pg_dump` gerar um script que requeira reconexões com o banco de dados quando for realizada a importação. Usualmente, um script de importação necessita reconectar várias vezes como usuários diferentes para especificar o dono original dos objetos. Esta opção é um instrumento bastante rudimentar, porque faz o `pg_dump` perder a informação sobre o dono, *a menos que* seja usada a opção `-X use-set-session-authorization`.

Uma das razões possíveis para não se desejar a reconexão durante a importação é o acesso ao banco de dados requerer intervenção manual (por exemplo, senhas).

Esta opção só faz sentido para o formato texto-puro. Para os outros formatos esta opção pode ser especificada ao se chamar o `pg_restore`.

-s
--schema-only

Exporta somente o esquema (definições dos dados), sem os dados.

-S *nome_do_usuário*
--superuser=*nome_do_usuário*

Os scripts e os arquivos de exportação gerados pelo `pg_dump` necessitam do privilégio de superusuário em certos casos, como desativar gatilhos ou definir o dono dos elementos do esquema. Esta opção especifica o nome do usuário a ser usado nestes casos.

-t *tabela*
--table=*tabela*

Exporta os dados da *tabela* apenas.

-v
--verbose

Especifica o modo verboso.

-x
--no-privileges
--no-acl

Impede gerar os privilégios de acessos (comandos GRANT/REVOKE).

-X use-set-session-authorization
--use-set-session-authorization

Normalmente, se o script (modo texto-puro) gerado pelo `pg_dump` necessita trocar o usuário corrente do banco de dados (por exemplo, para definir o dono correto do objeto) é usado o comando `\connect` do psql. Este comando na verdade abre uma nova conexão, o que pode requerer a intervenção manual (por exemplo, senhas). Se for usada a opção `-X use-set-session-authorization`, então o `pg_dump` vai usar o comando *SET SESSION AUTHORIZATION*. Embora produza o mesmo efeito,

requer que o usuário que for fazer a importação do banco de dados a partir do script gerado seja um superusuário. Esta opção substitui a opção `-R`.

Como o *SET SESSION AUTHORIZATION* é um comando SQL padrão, enquanto o `\connect` somente funciona no psql, esta opção também aumenta a portabilidade teórica do script gerado.

Esta opção só faz sentido para o formato texto-puro. Para os outros formatos esta opção pode ser especificada ao se chamar o `pg_restore`.

-Z *0..9*
--compress=*0..9*

Especifica o nível de compressão a ser usado nos arquivos com formatos que suportam compressão (atualmente somente o formato personalizado suporta compressão).

O `pg_dump` também aceita os seguintes argumentos de linha de comando para os parâmetros de conexão:

-h *hospedeiro*
--host=*hospedeiro*

Especifica o nome da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/), é considerado como sendo o diretório do soquete do domínio Unix.

-p *porta*
--port=*porta*

Especifica a porta Internet TCP/IP, ou o soquete do domínio local Unix, onde o servidor está aguardando as conexões. O padrão para o número da porta é 5432, ou o valor da variável de ambiente `PGPORT` (se estiver definida).

-U *nome_do_usuário*

Nome do usuário para se conectar.

-W

Força a solicitação da senha. Deve acontecer automaticamente se o servidor requerer autenticação por senha.

Diagnósticos

```
    Connection to database ’template1’ failed.
connectDBStart() -- connect() failed: No such file or directory
        Is the postmaster running locally
        and accepting connections on Unix socket ’/tmp/.s.PGSQL.5432’?
```

O pg_dump não pôde se conectar ao processo postmaster usando o computador e a porta especificada. Se esta mensagem for recebida, deve-se garantir que o postmaster está processando na máquina especificada e usando a porta especificada.

Note: O pg_dump executa internamente comandos SELECT. Se houver problema ao executar o pg_dump, deve-se ter certeza de poder consultar as informações no banco de dados usando, por exemplo, o psql.

Notas

Se na instalação houver alguma adição local ao banco de dados template1, deve-se ter o cuidado de restaurar a saída do pg_dump em um banco de dados realmente vazio; de outra forma podem acontecer erros devido à duplicidade de definições dos objetos adicionados. Para criar um banco de dados vazio, sem nenhuma adição local, deve-se fazê-lo partir do template0, e não do template1. Por exemplo:

```
CREATE DATABASE foo WITH TEMPLATE = template0;
```

O pg_dump possui algumas poucas limitações:

- Ao exportar uma única tabela, ou no formato texto-puro, o pg_dump não trata objetos grandes. Os objetos grandes devem ser exportados em sua inteireza usando um dos formatos binários de exportação.
- Ao exportar somente os dados, o pg_dump gera comandos para desativar os gatilhos das tabelas do usuário antes de inserir os dados, e comandos para reativá-los após os dados terem sido inseridos. Se a restauração for interrompida no meio, os catálogos do sistema podem ficar em um estado errado.

Exemplos

Para exportar um banco de dados:

```
$ pg_dump meu_bd > db.out
```

Para importar este banco de dados:

```
$ psql -d database -f db.out
```

Para exportar um banco de dados chamado meu_bd que contém objetos grandes para um arquivo tar:

```
$ pg_dump -Ft -b meu_bd > bd.tar
```

Para importar este banco de dados (com os objetos grandes) para um banco de dados existente chamado `novo_bd`:

```
$ pg_restore -d novo_bd bd.tar
```

Histórico

O utilitário `pg_dump` apareceu pela primeira vez no Postgres95 versão `0.02`. Os formatos de saída não-texto-puro foram introduzidos no PostgreSQL versão `7.1`.

Consulte também

pg_dumpall, pg_restore , psql, *Guia do Administrador do PostgreSQL*

pg_dumpall

Name

`pg_dumpall` — extrai todos os bancos de dados do PostgreSQL para um arquivo script

Synopsis

`pg_dumpall` [-c | --clean] [-g | --globals-only] [-h *hospedeiro*] [-p *porta*] [-U *nome_do_usuário*] [-W]

Descrição

O pg_dumpall é um utilitário para exportar (“dumping”) todos os bancos de dados do PostgreSQL para um arquivo script. O arquivo script contém comandos SQL que podem ser usados como entrada do psql para restaurar os bancos de dados. Exporta chamando pg_dump para cada banco de dados do PostgreSQL. O pg_dumpall também exporta objetos globais que são comuns a todos os bancos de dados (O pg_dump não salva estes objetos). Atualmente isto inclui informações sobre os usuários do banco de dados e os grupos.

Portanto, o pg_dumpall é uma solução integrada para realizar cópias de segurança dos bancos de dados. Mas observe a limitação: não pode exportar “objetos grandes”, porque o pg_dump não exporta estes objetos para arquivos texto. Se existirem bancos de dados contendo objetos grandes, estes devem ser exportados usando um dos modos não-texto do pg_dump.

Como o pg_dumpall lê as tabelas de todos os bancos de dados, muito provavelmente será necessário se conectar como um superusuário para poder gerar uma exportação completa. Também será necessário o privilégio de superusuário para executar o script salvo, para poder criar usuários e grupos, e para poder criar os bancos de dados.

O script SQL é escrito na saída padrão. Devem ser usados operadores na linha de comando para redirecioná-lo para um arquivo.

Opções

O pg_dumpall aceita os seguintes argumentos de linha de comando:

-c, --clean

Gera comandos para excluir (drop) os objetos do banco de dados antes de criá-los (Esta opção é sem utilidade prática, porque o script gerado espera criar os bancos de dados, e estes estarão sempre vazios após a criação).

-g, --globals-only

Somente exporta os objetos globais (usuários e grupos), não os bancos de dados.

-h *hospedeiro*

Especifica o nome da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/), é considerado como sendo o diretório do soquete do domínio Unix. O padrão é obtido da variável de ambiente `PGHOST`, se estiver definida, senão é tentada uma conexão pelo soquete do domínio Unix.

-p *porta*

O número da porta na qual o servidor está aguardando as conexões. O padrão é obtido da variável de ambiente PGPORT se estiver definida, ou do padrão da compilação.

-U *nome_do_usuário*

Nome do usuário para se conectar.

-W

Força a solicitação da senha. Deve acontecer automaticamente se o servidor requerer autenticação por senha.

Todos os outros parâmetros de linha de comando são passados para as chamadas subjacentes do pg_dump , sendo útil para controlar certos aspectos do formato da saída, mas algumas opções como -f, -t e *nome_bd* devem ser evitadas.

Exemplos

Para exportar todos bancos de dados:

```
$ pg_dumpall > db.out
```

Para importar estes bancos de dados deve-se usar, por exemplo:

```
$ psql -f db.out template1
```

(Não é importante em qual banco de dados se conectar porque o script criado pelo pg_dumpall contém os comandos apropriados para criar e conectar aos bancos de dados salvos).

Consulte também

pg_dump , psql. Veja também detalhes sobre as possíveis condições de erro.

pg_restore

Name

`pg_restore` — restaura um banco de dados do PostgreSQL a partir de um arquivo gerado pelo pg_dump

Synopsis

`pg_restore` [-a] [-c] [-C] [-d *nome_bd*] [-f *arquivo_de_saída*] [-F *formato*] [-i *índice*] [-l] [-L *arquivo_da_listagem*] [-N | -o | -r] [-O] [-P *nome_da_função*] [-R] [-s] [-S] [-t *tabela*] [-T *gatilho*] [-v] [-x] [-X *palavra_chave*] [-h *hospedeiro*] [-p *porta*] [-U *nome_do_usuário*] [-W] [*arquivo_de_exportação*]

Descrição

O `pg_restore` é um utilitário para restaurar um banco de dados do PostgreSQL a partir de um arquivo gerado pelo pg_dump em um dos formatos não-texto-puro. São executados os comandos necessários para criar novamente todos os tipos, funções, tabelas, índices, agregações e operadores definidos pelo usuário, assim como os dados das tabelas.

Os arquivos de exportação contêm informações para o `pg_restore` reconstruir o banco de dados, mas também permitem ao `pg_restore` selecionar o que deve ser restaurado, ou mesmo reordenar a restau-raração dos itens. Os arquivos de exportação são projetados para serem portáveis entre arquiteturas.

O `pg_restore` pode operar de dois modos: Se um nome de banco de dados for especificado, o arquivo de exportação é restaurado diretamente no banco de dados. Senão, um script contendo os comandos SQL necessários para reconstruir o banco de dados é criado (e escrito em um arquivo ou na saída padrão), semelhante aos scripts criados pelo `pg_dump` no formato texto-puro. Algumas das opções que controlam a criação do script são, portanto, análogas às opções do `pg_dump`.

Obviamente, o `pg_restore` não pode restaurar informações que não estejam presentes no arquivo de exportação; por exemplo, se o arquivo de exportação foi gerado usando a opção “exportar dados como `INSERT`”, o `pg_restore` não poderá importar os dados usando o comando `COPY`.

Opções

O `pg_restore` aceita os seguintes argumentos de linha de comando (As formas longas das opções estão disponíveis em algumas plataformas apenas).

nome_do_arquivo_exportado

Especifica a localização do arquivo de exportação a ser restaurado. Se não for especificado, a entrada padrão é usada.

-a
--data-only

Importa somente os dados, não o esquema (definições dos dados).

-c
--clean

Exclui (drop) os objetos do banco de dados antes de criá-los..

-C
--create

Cria o banco de dados antes de restaurá-lo (Quando esta opção está presente, o banco de dados designado por `-d` é usado apenas para executar o comando CREATE DATABASE inicial. Todos os dados são restaurados no banco de dados cujo nome aparece no arquivo de exportação).

-d *nome_bd*
--dbname=*nome_bd*

Conecta ao *nome_bd* e restaura diretamente no banco de dados. Os objetos grandes somente podem ser restaurados usando uma conexão direta ao banco de dados.

-f *arquivo_de_saída*
--file=*arquivo_de_saída*

Especifica o nome do arquivo contendo o script gerado, ou a listagem quando for utilizado com a opção `-l`. Por padrão a saída padrão.

-F *formato*
--format=*formato*

Especifica o formato do arquivo de exportação. Não é necessário especificar o formato, porque o `pg_restore` reconhece o formato automaticamente. Se for especificado, poderá ser um dos seguintes:

t

O arquivo de exportação está no formato `tar`. Este formato de arquivo de exportação permite reordenar e/ou excluir elementos do esquema durante a importação. Também permite limitar quais dados são recarregados durante a importação.

c

O arquivo de exportação está no formato personalizado do `pg_dump`. Este é o formato mais flexível porque permite a reordenação da importação dos dados e dos elementos do esquema. Este formato também é comprimido por padrão.

-i *índice*
--index=*índice*

Restaura a definição do índice para o *índice* especificado apenas.

-l
--list

Lista o conteúdo do arquivo de exportação. A saída deste comando pode ser usada com a opção `-L` para restringir e reordenar os itens que são restaurados.

-L `arquivo_da_listagem`
--use-list=`arquivo_da_listagem`

Restaura apenas os elementos presentes no `arquivo_da_listagem`, e na ordem em que aparecem neste arquivo. As linhas podem ser movidas e, também, podem virar comentário colocando-se um `;` no seu início.

-N
--orig-order

Restaura os itens na ordem original de exportação. Por padrão, o `pg_dump` irá exportar os itens em uma ordem conveniente para o `pg_dump`, e depois salvar o arquivo de exportação em uma ordem de OID modificada. Esta opção substitui a da ordem de OID.

-o
--oid-order

Restaura os itens na ordem de OID. Por padrão o `pg_dump` irá exportar exporta os itens em uma ordem conveniente para o `pg_dump`, e depois salvar o arquivo de exportação em uma ordem de OID modificada. Esta opção impõe a estrita ordem de OID.

-O
--no-owner

Impede qualquer tentativa de restaurar o dono original do objeto. O dono dos objetos será o usuário conectado ao banco de dados.

-P `nome_da_função`
--function=`nome_da_função`

Especifica o procedimento ou a função a ser restaurada.

-r
--rearrange

Restaura os itens na ordem modificada de OID. Por padrão, o `pg_dump` irá exportar os itens em uma ordem conveniente para o `pg_dump`, e depois salvar o arquivo de exportação em uma ordem de OID modificada. A maior parte dos objetos é restaurada na ordem de OID, mas alguns elementos (por exemplo, regras e índices) são restaurados no fim do processo sem respeitar os OIDs. Esta é a opção padrão.

-R
--no-reconnect

Durante a restauração do arquivo de exportação, o `pg_restore` usualmente necessita reconectar ao banco de dados vária vezes com nomes de usuário diferentes, para definir o dono correto dos objetos criados. Se isto não for desejável (por exemplo, se a intervenção manual for necessária para cada reconexão), esta opção proíbe o `pg_restore` requisitar reconexões (uma requisição de conexão em modo texto-puro, não conectado ao banco de dados, é feita emitindo o comando `\connect` do psql). Entretanto, esta opção é um instrumento bastante rudimentar, porque faz o `pg_restore` perder a informação sobre o dono, *a menos que* seja usada a opção `-X use-set-session-authorization`.

-s
--schema-only

Restaura somente o esquema (definições dos dados), sem os dados. Os valores das seqüências são substituídos.

-S *nome_do_usuário*
--superuser=*nome_do_usuário*

Especifica o nome do superusuário a ser usado para desativar os gatilhos e/ou definir o dono dos elementos do esquema. Por padrão, o `pg_restore` usa o nome do usuário corrente se este for um superusuário.

-t *tabela*
--table=*tabela*

Restaurar o esquema/dados da *tabela* apenas.

-T *gatilho*
--trigger=*gatilho*

Restaurar a definição do *gatilho* apenas.

-v
--verbose

Especifica o modo verboso.

-x
--no-privileges
--no-acl

Proíbe a restauração dos privilégios de acesso (comandos GRANT/REVOKE).

-X use-set-session-authorization
--use-set-session-authorization

Normalmente, se ao restaurar um arquivo de exportação for necessário trocar o usuário corrente do banco de dados (por exemplo, para definir o dono correto do objeto), uma nova conexão ao banco de dados deve ser aberta, o que poderá requerer intervenção manual (por exemplo, senhas). Se for usada a opção `-X use-set-session-authorization`, então o `pg_restore` vai usar o comando SET SESSION AUTHORIZATION. Embora produza o mesmo efeito, requer que o usuário que for fazer a importação do banco de dados a partir do arquivo de exportação gerado seja um superusuário. Esta opção substitui a opção `-R`.

O `pg_restore` também aceita os seguintes argumentos de linha de comando para os parâmetros de conexão:

-h *hospedeiro*
--host=*hospedeiro*

Especifica o nome da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/), é considerado como sendo o diretório do soquete do domínio Unix.

-p *porta*
--port=*porta*

Especifica a porta Internet TCP/IP, ou o soquete do domínio local Unix, onde o servidor está aguardando as conexões. O padrão para o número da porta é 5432, ou o valor da variável de ambiente `PGPORT` (se estiver definida).

-U *nome_do_usuário*

Nome do usuário para se conectar.

-W

Força a solicitação da senha. Deve acontecer automaticamente se o servidor requerer autenticação por senha.

Diagnósticos

```
Connection to database 'template1' failed.
connectDBStart() -- connect() failed: No such file or directory
        Is the postmaster running locally
        and accepting connections on Unix socket '/tmp/.s.PGSQL.5432'?
```

O `pg_restore` não pôde se conectar ao processo `postmaster` usando o computador e a porta especificada. Se esta mensagem for recebida, deve-se garantir que o servidor está processando na máquina especificada e usando a porta especificada. Se a instalação usa um sistema de autenticação, assegure-se de ter obtido as credenciais de autenticação necessárias.

Note: Quando uma conexão direta ao banco de dados é especificada através da opção -d, o `pg_restore` executa internamente os comandos SQL. Se houver problema ao executar o `pg_restore`, deve-se ter certeza de poder consultar as informações no banco de dados usando, por exemplo, o `psql`.

Notas

Se na instalação houver alguma adição local ao banco de dados template1, deve-se ter o cuidado de restaurar a saída do `pg_restore` em um banco de dados realmente vazio; de outra forma podem acontecer erros devido à duplicidade de definições dos objetos adicionados. Para criar um banco de dados vazio, sem nenhuma adição local, deve-se fazê-lo a partir do template0, e não do template1. Por exemplo:

```
CREATE DATABASE foo WITH TEMPLATE = template0;
```

As limitações do `pg_restore` estão descritas abaixo:

- Ao se restaurar os dados para tabelas pré-existentes, o `pg_restore` emite comandos para desativar os gatilhos das tabelas dos usuários antes de inserir os dados, e comandos para reativá-los após os dados terem sido inseridos. Se a restauração for interrompida no meio, os catálogos do sistema podem ficar em um estado errado.
- O `pg_restore` não restaura objetos grandes para uma única tabela. Se o arquivo de exportação contém objetos grandes, então todos os objetos grandes são restaurados.

Consulte a documentação do pg_dump para obter detalhes sobre as limitações do `pg_dump`.

Exemplos

Para exportar um banco de dados:

```
$ pg_dump meu_bd > bd.out
```

Para importar este banco de dados:

```
$ psql -d database -f bd.out
```

Para exportar um banco de dados chamado `meu_bd` que contém objetos grandes para um arquivo `tar`:

```
$ pg_dump -Ft -b meu_bd > bd.tar
```

Para importar este banco de dados (com os objetos grandes) para um banco de dados existente chamado `bd_novo`:

```
$ pg_restore -d bd_novo bd.tar
```

Para reordenar os itens do banco de dados, primeiro é necesário exportar a tabela de conteúdo (índice) do arquivo de exportação:

```
$ pg_restore -l arqexp.file > arqexp.list
```

O arquivo de listagem consiste do cabeçalho e de uma linha para cada item, como no exemplo abaixo:

```
;
; Archive created at Fri Jul 28 22:28:36 2000
;     dbname: birds
;     TOC Entries: 74
;     Compression: 0
;     Dump Version: 1.4-0
;     Format: CUSTOM
;
;
```

```
; Selected TOC Entries:
;
2; 145344 TABLE species postgres
3; 145344 ACL species
4; 145359 TABLE nt_header postgres
5; 145359 ACL nt_header
6; 145402 TABLE species_records postgres
7; 145402 ACL species_records
8; 145416 TABLE ss_old postgres
9; 145416 ACL ss_old
10; 145433 TABLE map_resolutions postgres
11; 145433 ACL map_resolutions
12; 145443 TABLE hs_old postgres
13; 145443 ACL hs_old
```

Ponto-e-vírgula são delimitadores de comentários, e os números no início das linhas referem-se aos identificadores interno do arquivo de exportação atribuídos a cada item.

As linhas do arquivo podem ser transformadas em comentário, excluídas e reordenadas. Por exemplo:

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

poderia ser usado como entrada para o `pg_restore` e somente restauraria os itens 10 e 6, nesta ordem.

```
$ pg_restore -L arqexp.list arqexp.file
```

Histórico

O utilitário `pg_restore` apareceu pela primera vez no PostgreSQL 7.1.

Consulte também

pg_dump , pg_dumpall, psql, *Guia do Administrador do PostgreSQL*

psql

Name

psql — terminal interativo do PostgreSQL

Synopsis

```
psql [ opções ] [ nome_bd [ nome_usuário ] ]
```

Sumário

O psql é um cliente do PostgreSQL em modo terminal. Permite digitar os comandos interativamente, enviá-los para o PostgreSQL e ver os resultados. Alternativamente, a entrada pode vir de um arquivo. Adicionalmente, possui um certo número de meta-comandos e diversas funcionalidades semelhantes às da `shell` para facilitar a criação de scripts e automatizar uma grande variedade de tarefas.

Descrição

Conexão a um banco de dados

O psql é um aplicativo cliente do PostgreSQL comum. Para se conectar a um banco de dados é necessário saber o nome do banco de dados, o nome da máquina e o número da porta do servidor, e o nome de usuário a ser usado para a conexão. O psql pode ser informado sobre estes parâmetros através das opções de linha de comando `-d`, `-h`, `-p` e `-U`, respectivamente. Se for encontrado um argumento que não pertença a nenhuma opção, este será interpretado como o nome do banco de dados (ou o nome do usuário, se o nome do banco de dados for fornecido). Nem todas estas opções são requeridas, os padrões se aplicam. Se for omitido o nome da máquina, então o psql se conecta através do soquete do domínio Unix ao servidor na máquina local. O número padrão para a porta é determinado em tempo de compilação. Desde que o servidor de banco de dados use o mesmo padrão, não será necessário especificar a porta na maioria dos casos. O nome de usuário padrão é o nome do usuário do Unix , como também é o nome do banco de dados padrão. Observe que não é possível se conectar a qualquer banco de dados com qualquer nome de usuário. O administrador de banco de dados informa as permissões de acesso concedidas. Para evitar a digitação, podem ser definidas as variáveis de ambiente `PGDATABASE`, `PGHOST`, `PGPORT` e `PGUSER` com os valores apropriados.

Se a conexão não puder ser estabelecida por algum motivo (por exemplo: privilégios insuficientes, o postmaster não está executando no servidor, etc...), o psql retorna uma mensagem de erro e termina.

Entrando com comandos

No modo normal de operação, o psql disponibiliza um `prompt` com o nome do banco de dados ao qual está conectado, seguido pela cadeia de caracteres `=>`. Por exemplo:

```
$ psql testdb
Welcome to psql, the PostgreSQL interactive terminal.
```

```
Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

testdb=>
```

No `prompt` o usuário pode digitar comandos SQL. Normalmente, as linhas de entrada são enviadas para o servidor quando o caractere ponto-e-vírgula, que termina o comando, é encontrado. Um caractere de fim-de-linha não termina um comando! Portanto os comandos podem ocupar várias linhas para clareza. Se o comando for enviado e executado sem erro, o resultado do comando será exibido na tela.

Sempre que um comando é executado, o psql também procura por eventos de notificação assíncronos gerados pelo *LISTEN* e *NOTIFY*.

Meta-comandos do psql

Qualquer coisa entrada no psql que comece por uma contrabarra (\) (não entre apóstrofos, `'`) é um meta-comando do psql que é processado pelo próprio psql. Estes comandos são o que fazem o psql interessante para a administração ou para a edição de scripts. Os meta-comandos são usualmente chamados de comandos de barra ou de contrabarra.

O formato do comando psql é a contrabarra, seguida imediatamente pelas letras do comando e depois pelos argumentos. Os argumentos são separados das letras do comando, e entre si, por qualquer número de caracteres de espaço.

Para incluir caracteres de espaço em um argumento deve-se colocá-los entre apóstrofos (`'`). Para incluir um apóstrofo neste tipo de argumento, deve-se precedê-lo por uma contrabarra. Qualquer coisa entre apóstrofos está sujeita às substituições no estilo C para o `\n` (nova-linha), `\t` (tabulação), `\`*`dígitos`*, `\0`*`dígitos`* e `\0x`*`dígitos`* (o caractere com o código decimal, octal ou hexadecimal informado).

Se o argumento (não entre apóstrofos) começar por dois-pontos (`:`), este será considerado como sendo uma variável e o valor desta variável será tomado como o argumento.

Os argumentos entre "crases" (`` ` ``) são considerados como sendo uma linha de comando a ser passada para a `shell`. A saída do comando (com o caractere de nova-linha final removido) é tomada como o argumento. As seqüências de escape (\) acima também se aplicam às crases.

Alguns comandos recebem como argumento o nome de um identificador SQL (como o nome de uma tabela). Estes argumentos seguem as regras de sintaxe do SQL com relação às aspas (`"`): um identificador que não esteja entre aspas é convertido para letras minúsculas. Para todos os outros comandos as aspas não são especiais, se tornando parte do argumento.

A leitura dos argumentos termina quando outra contrabarra (não entre apóstrofos) é encontrada. Esta é considerada como sendo o início de um novo meta-comando. A seqüência especial `\\` (duas contrabarras) marca o fim dos argumentos e prossegue analisando os comandos SQL, se existirem. Desta forma, os comandos SQL e psql podem ser livremente misturados em uma linha. Mas, em nenhum caso, os argumentos do meta-comando podem continuar além do fim da linha.

Os seguintes meta-comandos estão definidos:

`\a`

Se o formato atual de saída da tabela for desalinhado, muda para alinhado. Se não for desalinhado, define como desalinhado. Este comando é mantido para compatibilidade com as versões anteriores. Veja o `\pset` para uma solução geral.

`\cd` [*nome_do_diretório*]

Muda o diretório de trabalho corrente para *nome_do_diretório*. Sem argumento, muda para o diretório `home` do usuário corrente.

Tip: Para ver o diretório de trabalho corrente use `\!pwd`.

`\C` [*título*]

Define o título de qualquer tabela sendo exibida como resultado de uma consulta, ou remove a definição deste título. Este comando é equivalente ao `\pset title` *título* (O nome deste comando deriva de “caption” (título), porque anteriormente só era usado para definir o título de uma tabela em HTML).

`\connect` (ou `\c`) [*nome_bd* [*nome_usuário*]]

Estabelece a conexão com um novo banco de dados e/ou sob o mesmo nome de usuário. A conexão anterior é fechada. Se o *nome_bd* for `-` (hífen), então o nome do banco de dados corrente é assumido.

Se *nome_usuário* for omitido, o nome do usuário corrente é assumido.

Como regra especial, `\connect` sem nenhum argumento conecta ao banco de dados padrão como o usuário padrão (da mesma forma que aconteceria se o psql fosse executado sem argumentos).

Se a tentativa de conexão falhar (nome de usuário errado, acesso negado, etc...), a conexão anterior será mantida se, e somente se, o psql estiver no modo interativo. Ao executar um script não interativo, o processamento será imediatamente interrompido com um erro. Esta distinção foi escolhida por ser mais conveniente para o usuário que digita menos, e para garantir um mecanismo seguro que impeça os scripts atuarem acidentalmente no banco de dados errado.

`\copy` *table* [`with oids`] { `from` | `to` } *nome_do_arquivo* | stdin | stdout [`using delimiters` ’*caracteres*’] [`with null as` ’*cadeia_de_caracteres*’]

Executa uma cópia pelo cliente. Esta é uma operação que executa o comando SQL *COPY*, mas em vez do servidor ler ou escrever no arquivo especificado, e conseqüentemente ser necessário o acesso ao servidor e privilégios especiais de usuário, assim como estar limitado pelo sistema de arquivos acessível ao servidor, o psql lê ou escreve o arquivo e roteia os dados entre o servidor e o sistema de arquivos local.

A sintaxe do comando é semelhante à do comando SQL `COPY` (Consulte sua descrição para obter detalhes). Observe que, por causa disto, regras especiais de análise se aplicam ao comando `\copy`. Em particular, as regras de substituição de variável e de escapes de contrabarra não se aplicam.

Tip: Este modo de operação não é tão eficiente quanto o comando SQL `COPY`, porque todos os dados passam através da conexão IP cliente/servidor ou pelo soquete. Para uma grande quantidade de dados, o outro modo pode ser preferível.

Note: Observe a diferença de interpretação da `stdin` e da `stdout` entre as cópias feitas no cliente e no servidor: na cópia no cliente estes se referem sempre à entrada e a saída do psql. Na cópia pelo servidor, a `stdin` vem do mesmo lugar que o comando `COPY` veio (por exemplo, um script executado com a opção `-f`), e a `stdout` se refere à saída do comando (veja o meta-comando `\o` abaixo).

`\copyright`

Mostra os termos da distribuição e dos direitos autorais do PostgreSQL.

`\d` *`relação`*

Mostra todas as colunas da *`relação`* (que pode ser uma tabela, visão, índice ou seqüência), seus tipos e os atributos especiais como `NOT NULL` ou valor padrão, se houver. Se a relação for de fato uma tabela, todos os índices definidos, chaves primárias, restrições de unicidade e de verificação também são listadas. Se a relação for uma visão, a definição da visão também é mostrada.

A forma do comando `\d+` é idêntica, mas todos os comentários associados às colunas da tabela também são mostrados.

Note: Se `\d` for chamado sem nenhum argumento, torna-se equivalente ao `\dtvs` mostrando uma lista de todas as tabelas, visões e seqüências. Isto é puramente uma medida de conveniência.

`\da` [*`padrão`*]

Lista todas as funções de agregação disponíveis, juntamente com os tipos de dado com que operam. Se *`padrão`* (uma expressão regular) for especificado, somente as agregações correspondentes serão exibidas.

`\dd` [*`objeto`*]

Mostra a descrição do *`objeto`* (que pode ser uma expressão regular), ou de todos os objetos se nenhum argumento for fornecido ("Objeto" compreende agregações, funções, operadores, tipos, relações [tabelas, visões, índices, seqüências, objetos grandes], regras e gatilhos). Por exemplo:

```
=> \dd version
              Object descriptions
  Name   |   What   |        Description
---------+----------+---------------------------
 version | function | PostgreSQL version string
(1 row)
```

As descrições dos objetos podem ser geradas pelo comando SQL `COMMENT ON`.

Note: O PostgreSQL armazena a descrição dos objetos na tabela do sistema `pg_description`.

`\df [ padrão ]`

Lista as funções disponíveis, juntamente com o tipo de dado de seus argumentos e do valor retornado. Se *padrão* (uma expressão regular) for especificado, somente as funções correspondentes serão mostradas. Se a forma `\df+` for usada, informações adicionais sobre cada função, incluindo a linguagem e a descrição, são mostradas.

`\distvS [ padrão ]`

Este não é o verdadeiro nome do comando: As letras i, s, t, v, S correspondem ao índice, seqüência, tabela, visão e tabela do sistema, respectivamente. Pode-se especificar qualquer um deles, ou todos eles, em qualquer ordem, para obter uma listagem dos mesmos juntamente com a informação sobre quem é o dono.

Se o *padrão* for especificado, é tomado como uma expressão regular que restringe a listagem aos objetos com nomes correspondentes. Se for apensado o caractere "+" ao nome do comando, cada objeto é listado juntamente com a sua descrição, se houver.

`\dl`

Este é um sinônimo para `\lo_list`, que mostra a lista dos objetos grandes.

`\do [ nome ]`

Lista os operadores disponíveis juntamente com o tipo de seus operandos e do valor retornado. Se *nome* for especificado, somente operadores com este nome serão mostrados.

`\dp` [*padrão*]

Este é um sinônimo para `\z` que foi incluído devido ao seu maior valor mnemônico ("display permissions") (mostra permissões).

`\dT [ padrão ]`

Lista todos os tipos de dado, ou somente àqueles que correspondem ao *padrão*. A forma do comando `\dT+` mostra informações extras.

`\du [ padrão ]`

Lista todos os usuários configurados, ou somente àqueles que correspondem ao *padrão*.

`\edit` (ou `\e`) [*nome_do_arquivo*]

Se o *nome_do_arquivo* for especificado, o arquivo é editado; após o fim da execução do editor, o conteúdo do arquivo é copiado para o `buffer` de comando. Se nenhum argumento for fornecido, o `buffer` de comando corrente é copiado para um arquivo temporário, que é editado de maneira idêntica.

O novo `buffer` de comando é então analisado novamente de acordo com as regras normais do psql, onde todo o `buffer` é tratado como sendo uma única linha (Portanto, não podem ser gerados scripts dessa maneira. Use o comando `\i` para fazer isso). Assim sendo, se o comando terminar por (ou contiver) um ponto-e-vírgula será executado imediatamente, senão apenas permanece aguardando no `buffer` de comando.

> **Tip:** O psql procura nas variáveis de ambiente `PSQL_EDITOR`, `EDITOR` e `VISUAL` (nesta ordem) o editor a ser usado. Se nenhuma delas estiver definida, `/bin/vi` é usado.

`\echo` *`texto`* [...]

Envia os argumentos para a saída padrão, separados por um espaço e seguido por um caractere de nova-linha, podendo ser útil para intercalar informações na saída dos scripts. Por exemplo:

```
=> \echo ‘date‘
Tue Oct 26 21:40:57 CEST 1999
```

Se o primeiro argumento for `-n` (não entre apóstrofos) o caractere de nova-linha final não é gerado.

> **Tip:** Se for usado o comando `\o` para redirecionar a saída dos comandos, talvez se deseje utilizar `\qecho` no lugar deste comando.

`\encoding` [*`codificação`*]

Define a codificação do cliente, se estiver sendo utilizada a codificação multibyte. Sem argumento, este comando mostra a codificação corrente.

`\f` [*`caracteres`*]

Define o separador de campos para a saída de comando não alinhada. O padrão é a barra vertical (`|`). Consulte também o `\pset` para ver uma forma genérica de definir as opções de saída.

`\g` [{ *`nome_do_arquivo`* | `|`*`comando`* }]

Envia o `buffer` de entrada de comando corrente para o servidor e, opcionalmente, salva a saída em *`nome_do_arquivo`*, ou envia a saída para uma outra `shell` do Unix para executar o *`comando`*. Um `\g` puro e simples é virtualmente igual ao ponto-e-vírgula. Um `\g` com argumento é uma alternativa “expressa” para o comando `\o`.

`\help` (ou `\h`) [*`comando`*]

Fornece ajuda para a sintaxe do comando SQL especificado. Se o *`comando`* não for especificado, então o psql irá listar todos os comandos para os quais a ajuda de sintaxe está disponível. Se o *`comando`* for um asterisco (“*”), então é mostrada a ajuda de sintaxe para todos os comandos SQL.

> **Note:** Para simplificar a digitação, os comandos constituídos por várias palavras não necessitam estar entre apóstrofos. Portanto, pode-se digitar **`\help alter table`**.

`\H`

Ativa o formato HTML de saída da consulta. Se o formato HTML já estiver ativado, retorna ao formato de texto alinhado padrão. Este comando existe por compatibilidade e conveniência, mas veja no `\pset` outras definições de opções de saída.

`\i` *`nome_do_arquivo`*

Ler a entrada no arquivo *`nome_do_arquivo`*, e executar como se tivesse sido digitada pelo teclado.

Note: Se for desejado ver as linhas na tela enquanto estas são lidas deve-se definir a variável `ECHO` como `all`.

`\l` (ou `\list`)

Lista todos os bancos de dados do servidor, assim como seus donos. Deve-se apensar o caractere "+" ao nome do comando para listar as descrições dos bancos de dados também. Se o PostgreSQL foi compilado com suporte à codificação multibyte, o esquema de codificação de cada banco de dados também é mostrado.

`\lo_export` *`loid`* *`nome_do_arquivo`*

Lê do banco de dados o objeto grande com OID igual a *`loid`*, e escreve em *`nome_do_arquivo`*. Observe que isto é sutilmente diferente da função do servidor `lo_export`, que atua com a permissão do usuário como o qual o servidor de banco de dados processa, e no sistema de arquivos do servidor.

Tip: Use `\lo_list` para descobrir os OIDs dos objetos grandes.

Note: Veja a descrição da variável `LO_TRANSACTION` para obter informações importantes com relação a todas as operações com objetos grandes.

`\lo_import` *`nome_do_arquivo`* [*`comentário`*]

Armazena o arquivo em um "objeto grande" do PostgreSQL. Opcionalmente, associa o comentário fornecido com o objeto. Exemplo:

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'uma fotografia minha'
lo_import 152801
```

A resposta indica que o objeto grande recebeu o identificador de objeto 152801, que deve ser lembrado para acessar o objeto novamente. Por esta razão, recomenda-se associar sempre um comentário inteligível a todos os objetos. Estes podem ser vistos através do comando `\lo_list`.

Observe que este comando é sutilmente diferente da função `lo_import` do servidor, porque atua como o usuário local no sistema de arquivos local, em vez do usuário e sistema de arquivos do servidor.

Note: Veja a descrição da variável `LO_TRANSACTION` para obter informações importantes com relação a todas as operações com objetos grandes.

`\lo_list`

Mostra a lista contendo todos os "objetos grandes" do PostgreSQL armazenados atualmente no banco de dados, juntamente com os comentários fornecidos para os mesmos.

`\lo_unlink` *`loid`*

Exclui do banco de dados o objeto grande com o OID igual a *`loid`*.

Tip: Use `\lo_list` para descobrir os OIDs dos objetos grandes.

Note: Veja a descrição da variável `LO_TRANSACTION` para obter informações importantes com relação a todas as operações com objetos grandes.

`\o` [{*nome_do_arquivo* | |*comando*}]

Salva os resultados das próximas consultas no arquivo *nome_do_arquivo*, ou envia os próximos resultados para uma outra `shell` do Unix para executar o *comando*. Se nenhum argumento for especificado, a saída da consulta será enviada para `stdout`.

Os “resultados das consultas” incluem todas as tabelas, respostas dos comandos e as notificações obtidas do servidor de banco de dados, assim como a saída dos vários comandos de contrabarra que consultam o banco de dados (como o `\d`), mas não as mensagens de erro.

Tip: Para intercalar texto entre os resultados das consultas usa-se `\qecho`.

`\p`

Envia o `buffer` de comando corrente para a saída padrão.

`\pset` *parâmetro* [*valor*]

Este comando define opções que afetam a saída das tabelas de resultado das consultas. O *parâmetro* indica qual opção será definida. A semântica do *valor* depende do parâmetro.

As opções ajustáveis de exibição são:

`format`

Define o formato de saída como `unaligned` (não alinhado), `aligned` (alinhado), `html` ou `latex`. Abreviações únicas são permitidas (O que equivale a dizer que uma letra basta).

O modo “Unaligned” escreve todos os campos da tupla em uma linha, separados pelo separador de campos ativo no momento. Pretende-se com isso poder criar uma saída que sirva de entrada para outro programa (separada por tabulação, por vírgula, etc...). O modo “Aligned” é a saída de texto padrão, inteligível e agradavelmente formatada. Os modos “HTML” e “LaTeX” produzem tabelas feitas para serem incluídas em documentos usando a linguagem de marcação correspondente. Não são documentos completos! (Isto não é tão problemático no HTML, mas no LaTeX deve haver um invólucro completo do documento).

`border`

O segundo argumento deve ser um número. Em geral, quanto maior o número mais bordas e linhas a tabela terá, mas isto depende do formato em particular. No modo HTML será traduzido diretamente para o atributo `border=...`, nos outros modos somente os valores 0 (sem borda), 1 (linhas divisórias internas) e 2 (moldura da tabela) fazem sentido.

`expanded` (ou `x`)

Alterna entre os formatos regular e expandido. Quando o formato expandido é ativado todas as saídas possuem duas colunas, ficando o nome do campo à esquerda e o valor à direita. Este modo é útil quando os dados não cabem na tela no modo normal “horizontal”.

O modo expandido é suportado por todos os quatro modos de saída.

`null`

O segundo argumento é a cadeia de caracteres a ser impressa sempre que o campo for nulo. O padrão é não imprimir nada, o que pode ser facilmente confundido com, por exemplo, uma cadeia de caracteres vazia. Portanto, pode-se preferir escrever `\pset null '(nulo)'`.

`fieldsep`

Especifica o separador de campos a ser utilizado no modo de saída não alinhado. Desta forma pode-se criar, por exemplo, uma saída separada por tabulação, por vírgula ou por um outro caractere, conforme a preferência do programa. Para definir o caractere de tabulação como separador de campo deve-se usar `\pset fieldsep '\t'`. O separador de campos padrão é o `'|'` (a barra vertical, ou o símbolo do “pipe”).

`footer`

Alterna a exibição do rodapé padrão `(x linhas)`.

`recordsep`

Especifica o separador de registro (linha) a ser usado no modo de saída não alinhado. O padrão é o caractere de nova-linha.

`tuples_only` (ou `t`)

Alterna entre exibir somente as tuplas e exibir tudo. O modo exibir tudo pode mostrar informações adicionais como os cabeçalhos das colunas, títulos e vários rodapés. No modo tuplas-apenas somente os dados da tabela são mostrados.

`title` [*`texto`*]

Define o título para as próximas tabelas a serem impressas. Pode ser usado para colocar textos descritivos na saída. Se nenhum argumento for fornecido, a definição é removida.

> **Note:** Anteriormente só afetava o modo HTML. Atualmente pode ser definido título para qualquer formato de saída.

`tableattr` (ou `T`) [*`texto`*]

Permite especificar qualquer atributo a ser colocado dentro do marcador `table` do HTML. Estes atributos podem ser, por exemplo, `cellpadding` ou `bgcolor`. Observe que, provavelmente, não vai se desejar especificar `border` aqui, porque isto já é tratado pelo `\pset border`.

`pager`

Alterna o paginador usado para mostrar a saída da tabela. Se a variável de ambiente `PAGER` estiver definida, a saída é enviada para o programa especificado, senão o `more` é utilizado.

De qualquer maneira, o psql somente usa o paginador se julgar adequado, significando que, entre outras coisas, a saída é para um terminal e que a tabela normalmente não caberia na tela. Devido

a natureza modular das rotinas de impressão, não é sempre possível predizer o número de linhas que serão realmente impressas. Por esta razão, o psql pode parecer não muito discriminativo sobre quando usar ou não o paginador.

Ilustrações mostrando como parecem estes formatos diferentes podem ser vistas na seção *Exemplos*.

Tip: Existem vários comandos abreviados para o `\pset`. Veja `\a`, `\C`, `\H`, `\t`, `\T` e `\x`.

Note: Atualmente é errado chamar o `\pset` sem argumentos. No futuro, esta chamada deverá mostrar o status corrente de todas as opções de impressão.

`\q`

Sair do programa psql.

`\qecho` *`texto`* [...]

Este comando é idêntico ao `\echo`, exceto que toda a saída é escrita no canal de saída de consulta, definido pelo `\o`.

`\r`

Restaura (limpa) o `buffer` de comando.

`\s` [*`nome_do_arquivo`*]

Exibe ou salva o histórico da linha de comando em *`nome_do_arquivo`*. Se *`nome_do_arquivo`* for omitido, o histórico é enviado para a saída padrão. Esta opção somente estará disponível se o psql estiver configurado para usar a biblioteca de histórico GNU.

Note: Na versão corrente não é mais necessário salvar o histórico dos comandos, porque isto é feito, automaticamente, ao término do programa. O histórico também é carregado, automaticamente, toda vez que o psql inicia.

`\set` [*`nome`* [*`valor`* [...]]]

Define a variável interna *`nome`* com o *`valor`* ou, se mais de um valor for fornecido, com a concatenação de todos eles. Se o segundo valor não for fornecido, a variável é definida sem valor. Para remover a definição da variável deve-se usar o comando `\unset`.

Nomes válidos de variáveis podem conter letras, dígitos e sublinhados (_). Veja a seção sobre as variáveis do psql para obter detalhes.

Embora possa ser definida qualquer variável, para fazer qualquer coisa que se deseje, o psql trata diversas variáveis como sendo especiais. Elas estão documentadas na seção sobre variáveis.

Note: Este comando é totalmente distinto do comando SQL *SET*.

`\t`

Alterna a exibição do cabeçalho contendo o nome das colunas e do rodapé contendo o número de linhas. Este comando é equivalente ao `\pset tuples_only`, sendo fornecido por conveniência.

`\T` *`opções_de_tabela`*

Permite especificar opções a serem colocadas na `tag table` no modo de saída tabular HTML. Este comando é equivalente ao `\pset tableattr` *`opções_de_tabela`*.

`\w` {*`nome_do_arquivo`* | **|***`comando`*}

Escreve o `buffer` de comando corrente no arquivo *`nome_do_arquivo`*, ou envia para o comando Unix *`comando`* através de um `pipe`.

`\x`

Alterna o modo estendido de formato de linha. Sendo assim, é equivalente ao `\pset expanded`.

`\z` [*`padrão`*]

Produz uma lista contendo todas as tabelas do banco de dados, juntamente com suas permissões de acesso. Se um argumento for fornecido, este será considerado como sendo uma expressão regular, limitando a lista às tabelas correspondentes.

```
test=> \z
Access permissions for database "test"
   Relation   |           Access permissions
--------------+-------------------------------------
 minha_tabela | {"=r","joe=arwR", "group staff=ar"}
(1 row )
```

Deve ser lido da seguinte maneira:

- `"=r"`: `PUBLIC` possui permissão de leitura (`SELECT`) na tabela.
- `"joe=arwR"`: O usuário `joe` possui permissão para ler, escrever (`UPDATE`, `DELETE`), “apensar” (`INSERT`), e criar regras na tabela.
- `"group staff=ar"`: O grupo `staff` possui permissão para `SELECT` e `INSERT`.

Os comandos GRANT e REVOKE são usados para definir as permissões de acesso.

`\!` [*`comando`*]

Abre outra `shell` do Unix, ou executa o comando Unix *`comando`*. Os comandos deixam de ser interpretados e são enviados para a `shell` como estão.

`\?`

Obtém informação de ajuda para os comandos de contrabarra (“\”).

Opções de linha de comando

Caso esteja configurado adequadamente, o psql trata tanto as opções curtas no estilo Unix padrão, quanto as opções longas no estilo GNU. Estas últimas não estão disponíveis em todos os sistemas operacionais.

-a, --echo-all

Exibe todas as linhas na tela ao serem lidas. É mais útil para o processamento de scripts do que no modo interativo. Equivale a definir a variável `ECHO` como `all`.

-A, --no-align

Muda para o modo de saída não alinhado (Senão, o modo de saída padrão será o alinhado).

-c, --command *`comando`*

Especifica que o psql deve executar a cadeia de caracteres *`comando`*, e depois terminar. Útil para scripts.

O *`comando`* deve ser uma cadeia de caracteres que possa ser integralmente analisada pelo servidor (ou seja, não contém nenhuma funcionalidade específica do psql), ou ser um único comando de contrabarra. Portanto, não podem ser misturados comandos SQL com meta-comandos do psql. Para que isto seja obtido, pode-se enviar a cadeia de caracteres para o psql conforme mostrado a seguir: `echo "\x \\ select * from foo;" | psql`.

-d, --dbname *`nome_bd`*

Especifica o nome do banco de dados a se conectar. Equivale a especificar o *`nome_bd`* como o primeiro argumento não-opção da linha de comando.

-e, --echo-queries

Mostra todos os comandos enviados para o servidor. Equivale a definir a variável `ECHO` como `queries`.

-E, --echo-hidden

Exibe o comando real gerado pelo \d e por outros comandos de contrabarra. Pode ser usado caso se deseje incluir uma funcionalidade similar nos próprios programas. Equivale a definir a variável `ECHO_HIDDEN` a partir do psql.

-f, --file *`nome_do_arquivo`*

Usa o arquivo *`nome_do_arquivo`* como origem dos comandos, em vez de ler os comandos interativamente. Após o arquivo ser processado, o psql termina. Sob muitos aspectos equivale ao comando interno `\i`.

Se o *`nome_do_arquivo`* for `-` (hífen) a entrada padrão é lida.

O uso desta opção é sutilmente diferente de escrever `psql <` *`nome_do_arquivo`*. De uma maneira geral as duas fazem o esperado, mas o uso de `-f` ativa algumas funcionalidades úteis, como as mensagens de erro contendo o número da linha. Ao se usar esta opção também existe uma pequena chance de reduzir a sobrecarga da inicialização. Por outro lado, utilizando o redirecionamento da entrada (em teoria) garante produzir exatamente a mesma saída que seria produzida se tudo tivesse sido digitado manualmente.

-F, --field-separator *`separador`*

Usa *`separador`* como o separador de campos. Equivalente ao `\pset fieldsep` ou ao `\f`.

-h, --host *hostname*

Especifica o nome da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/), é usado como sendo o diretório do soquete do domínio Unix.

-H, --html

Ativa a saída tabular HTML. Equivalente ao `\pset format html` ou ao comando `\H`.

-l, --list

Lista todos bancos de dados disponíveis e depois termina. Outras opções que não forem de conexão são ignoradas. Semelhante ao comando interno `\list`.

-o, --output *nome_do_arquivo*

Envia a saída de todos os comandos para o arquivo *nome_do_arquivo*. Equivalente ao comando `\o`.

-p, --port *porta*

Especifica a porta TCP/IP ou, por omissão, o soquete do domínio local Unix, onde o postmaster está aguardando as conexões. Por padrão o valor da variável de ambiente `PGPORT` ou, se não estiver definida, a porta especificada durante a compilação, usualmente 5432.

-P, --pset *atribuição*

Permite especificar opções de impressão no estilo `\pset` pela linha de comando. Observe que aqui o nome e o valor devem estar separados pelo sinal de igual em vez de espaço. Portanto, para definir o formato de saída como LaTeX, deve-se escrever `-P format=latex`.

-q

Especifica que o psql deve trabalhar em silêncio. Por padrão, são exibidas mensagens de boas-vindas e diversas outras mensagens informativas. Se esta opção for usada, nada disso acontece. É útil juntamente com a opção `-c`. Dentro do psql pode-se também definir a variável `QUIET` para obter o mesmo efeito.

-R, --record-separator *separador*

Usa o *separador* como separador de registros. Equivalente ao comando `\pset recordsep`.

-s, --single-step

Executa no modo passo-único, significando que será solicitada uma confirmação antes de cada comando ser enviado para o servidor, com a opção de cancelar a execução. Usado na depuração de scripts.

-S, --single-line

Executa no modo linha-única, onde o caractere de nova-linha termina o comando, como o ponto-e-vírgula faz.

> **Note:** Este modo é fornecido para àqueles que insistem em usá-lo, mas não se encoraja a sua utilização. Em particular, se forem misturados comandos SQL e meta-comandos na mesma linha, a ordem de execução nem sempre vai ser clara para o usuário inexperiente.

-t, --tuples-only

Desativa a impressão dos nomes das colunas e do rodapé com o número de linhas do resultado, etc... É totalmente equivalente ao meta-comando `\t`.

-T, --table-attr *`opções_de_tabela`*

Permite especificar opções a serem colocadas dentro da `tag table` do HTML. Veja `\pset` para obter detalhes.

-u

Faz com que o psql solicite o nome do usuário e a senha antes de se conectar ao banco de dados.

Esta opção está obsoleta, sendo conceitualmente errada (Solicitar um nome de usuário não por padrão e solicitar uma senha porque o servidor requer são duas coisas realmente diferentes). Encoraja-se o uso das opções `-U` e `-W` em seu lugar.

-U, --username *`nome_do_usuário`*

Conecta-se as banco de dados como o usuário *`nome_do_usuário`* em vez do padrão (É necessário ter permissão para fazê-lo, é claro).

-v, --variable, --set *`atribuição`*

Realiza a atribuição de variável, como o comando interno `\set`. Observe que é necessário separar o nome e o valor, se houver, por um sinal de igual na linha de comando. Para remover a definição de uma variável omite-se sinal de igual. Para definir uma variável sem um valor, usa-se o sinal de igual mas omite-se o valor. Estas atribuições são realizadas durante um estágio bem no princípio da inicialização, portanto as variáveis reservadas para finalidades internas devem ser sobrescritas mais tarde.

-V, --version

Mostra a versão do psql.

-W, --password

Requer que o psql solicite a senha antes de se conectar ao banco de dados. Esta continuará definida por toda a sessão, mesmo que a conexão ao banco de dados seja mudada com o meta-comando `\connect`.

Na versão corrente o psql solicita, automaticamente, a senha sempre que o servidor requerer autenticação por senha. Uma vez que atualmente isto se baseia em um `hack`, o reconhecimento automático pode falhar misteriosamente, e por isso existe esta opção para forçar a solicitação. Se a solicitação da senha não for feita, e se o servidor requerer autenticação por senha, a tentativa de conexão vai falhar.

-x, --expanded

Ativa o modo formato de linha estendido. Equivalente ao comando `\x`.

-X, --no-psqlrc

Não lê o arquivo de inicialização `~/.psqlrc`.

-?, --help

Mostra a ajuda para os argumentos de linha de comando do psql.

Funcionalidades avançadas

Variáveis

O psql fornece uma funcionalidade de substituição de variáveis similar às dos comandos comuns da `shell` do Unix. Esta funcionalidade é nova e não muito sofisticada ainda, mas existem planos para expandi-la no futuro. As variáveis são simplesmente um par nome/valor, onde o valor pode ser uma cadeia de caracteres de qualquer comprimento. Para definir uma variável usa-se o meta-comando do psql `\set`:

```
testdb=> \set foo bar
```

define a variável "foo" com o valor "bar". Para usar o conteúdo da variável deve-se preceder o nome por dois-pontos (`:`), e usá-la como argumento de qualquer comando de contrabarra:

```
testdb=> \echo :foo
bar
```

> **Note:** Os argumentos do `\set` estão sujeitos às mesmas regras de substituição de qualquer outro comando. Portanto, pode-se construir referências interessantes como `\set :foo 'something'` e obter "soft links" ou "variable variables" do Perl e do PHP, respectivamente. Desafortunadamente (ou afortunadamente?), não existe nenhuma maneira de se fazer qualquer coisa útil com estas construções. Por outro lado, `\set bar :foo` é uma forma perfeitamente válida de se copiar uma variável.

Se `\set` for chamado sem um segundo argumento, a variável é simplesmente definida, mas não possui valor. Para remover a definição (ou excluir) a variável, usa-se o comando `\unset`.

Os nomes das variáveis internas do psql podem consistir de letras, números e sublinhados em qualquer ordem e em qualquer número deles. Algumas variáveis regulares possuem tratamento especial pelo psql. Elas indicam certas definições de opções que podem ser mudadas em tempo de execução alterando-se o valor da variável, ou representam algum estado do aplicativo. Embora seja possível usar estas variáveis para qualquer outra finalidade, isto não é recomendado, uma vez que o comportamento do programa pode se tornar muito estranho e muito rapidamente. Por convenção, todas as variáveis com tratamento especial possuem todas as letras maiúsculas (e possivelmente números e sublinhados). Para garantir a máxima compatibilidade no futuro, evite estas variáveis. Abaixo segue a lista de todas as variáveis com tratamento especial.

`DBNAME`

O nome do banco de dados conectado correntemente. É definida toda vez que se conecta a um banco de dados (inclusive na inicialização), mas a definição pode ser removida.

`ECHO`

Se for definida como "`all`", todas as linhas entradas, ou do script, são escritas na saída padrão antes de serem analisadas ou executadas. Para especificar no início do programa usa-se a chave `-a`. Se for definido como "`queries`", o psql exibe todos os comandos antes de enviá-los para o servidor. A opção para isto é `-e`.

`ECHO_HIDDEN`

Quando esta variável está definida e um comando de contrabarra consulta o banco de dados, a consulta é mostrada previamente. Desta forma, pode-se estudar o PostgreSQL internamente e fornecer funcionalidades semelhantes nos próprios programas. Se a variável for definida como “noexec”, a consulta é apenas exibida sem ser enviada para o servidor para executar.

`ENCODING`

A codificação multibyte corrente do cliente. Se não estiver configurado para usar caracteres multibyte, esta variável irá conter sempre “SQL_ASCII”.

`HISTCONTROL`

Se esta variável estiver definida como `ignorespace`, as linhas que começam por espaço não são salvas na lista de histórico. Se estiver definida como `ignoredups`, as linhas idênticas à linha anterior do histórico não são salvas. O valor `ignoreboth` combina estas duas opções. Se não estiver definida, ou se estiver definida com um valor diferente destes acima, todas as linhas lidas no modo interativo são salvas na lista de histórico.

> **Note:** Esta funcionalidade foi plagiada do bash.

`HISTSIZE`

O número de comandos a serem armazenados no histórico de comandos. O valor padrão é 500.

> **Note:** Esta funcionalidade foi plagiada do bash.

`HOST`

O hospedeiro do servidor de banco de dados ao qual se está conectado. É definida toda vez que se conecta a um banco de dados (inclusive na inicialização), mas a definição não pode ser removida.

`IGNOREEOF`

Se não estiver definida, o envio de um caractere EOF (usualmente Control-D) para uma sessão interativa do psql termina o aplicativo. Se estiver definida com um valor numérico, este número de caracteres EOF são ignorados antes que o aplicativo termine. Se a variável estiver definida, mas não tiver um valor numérico, o padrão é 10.

> **Note:** Esta funcionalidade foi plagiada do bash.

`LASTOID`

O valor do último `oid` afetado, retornado por um comando `INSERT` ou `lo_insert`. Esta variável somente é garantida como válida até que o resultado do próximo comando SQL tenha sido mostrado.

`LO_TRANSACTION`

Se for usada a interface de objeto grande do PostgreSQL para armazenar os dados que não cabem em uma tupla, todas as operações devem estar contidas em um bloco de transação (Consulte a doc-

umentação da interface de objetos grandes para obter mais informações). Como o psql não tem maneira de saber se existe uma transação em andamento quando é chamado um de seus comandos internos (`\lo_export`, `\lo_import`, `\lo_unlink`) este precisa tomar alguma decisão arbitrária. Esta ação pode ser desfazer (`roll back`) alguma transação que esteja em andamento, efetivar esta transação (`commit`), ou não fazer nada. Neste último caso deve ser fornecido o bloco `BEGIN TRANSACTION`/`COMMIT`, ou o resultado não será previsível (geralmente resultando em que a ação desejada não seja realizada em nenhum caso).

Para escolher o que se deseja fazer deve-se definir esta variável como “rollback”, “commit” ou “nothing”. O padrão é desfazer (`roll back`) a transação. Se for desejado carregar apenas um ou poucos objetos está tudo bem. Entretanto, se a intenção for transferir muitos objetos grandes, é aconselhável fornecer um bloco de transação explícito em torno dos comandos.

`ON_ERROR_STOP`

Por padrão, se um script não-interativo encontrar um erro, como um comando SQL ou um meta-comando mal formado, o processamento continua. Este tem sido o comportamento tradicional do psql, mas algumas vezes não é o desejado. Se esta variável estiver definida, o processamento do script vai terminar imediatamente. Se o script for chamado por outro script este vai terminar da mesma maneira. Se o script externo não foi chamado por uma sessão interativa do psql, mas usando a opção `-f`, o psql retorna um código de erro 3, para distinguir este caso das condições de erro fatal (código de erro 1).

`PORT`

A porta do servidor de banco de dados que se está conectado. Definida toda vez que se conecta ao banco de dados (inclusive na inicialização), mas a definição pode ser removida.

`PROMPT1`, `PROMPT2`, `PROMPT3`

Especificam como os `prompts` emitidos pelo psql devem se parecer. Veja “*Prompt*” abaixo.

`QUIET`

Esta variável é equivalente à opção de linha de comando `-q`. Provavelmente não é muito útil no modo interativo.

`SINGLELINE`

Esta variável é definida pela opção de linha de comando `-S`. Pode ser redefinida ou ter a definição removida em tempo de execução.

`SINGLESTEP`

Esta variável é equivalente à opção de linha de comando `-s`.

`USER`

O usuário do banco de dados que se está conectado. Definida toda vez que se conecta ao banco de dados (inclusive na inicialização), mas a definição pode ser removida.

Interpolação SQL

Uma funcionalidade bastante prática das variáveis do psql é que podem ser substituídas (“interpoladas”) dentro de comandos regulares do SQL. A sintaxe para isto é novamente prefixar a variável com dois-pontos (:).

```
testdb=> \set foo ’minha_tabela’
testdb=> SELECT * FROM :foo;
```

faria então a consulta à tabela minha_tabela. O valor da variável é copiado literalmente podendo, portanto, conter apóstrofos não balanceados ou comandos de contrabarra. Deve-se ter certeza que faz sentido onde é colocada. A interpolação de variáveis não é realizada dentro de entidades SQL entre apóstrofos.

Uma aplicação comum desta funcionalidade é para fazer referência nos comandos seguintes ao último OID inserido, para construir um cenário de chave estrangeira. Outro uso possível deste mecanismo é copiar o conteúdo de um arquivo para um campo. Primeiro deve-se carregar o arquivo na variável e depois proceder coforme mostrado.

```
testdb=> \set content ’\” ‘cat meu_arquivo.txt‘ ’\”
testdb=> INSERT INTO minha_tabela VALUES (:content);
```

Um problema possível com esta abordagem é que o meu_arquivo.txt pode conter apóstrofos, que devem ser precedidos por uma contrabarra para que não causem erro de sintaxe quando a inserção for processada, o que poderia ser feito através do programa sed:

```
testdb=> \set content ’\” ‘sed -e "s/’/\\\\\\’/g" < meu_arquivo.txt‘ ’\”
```

Observe o número correto de contabarras (6)! Isto pode ser visto desta maneira: Após o psql ter analisado esta linha, vai enviar `sed -e "s/’/\\\’/g" < meu_arquivo.txt` para a shell. A shell fará suas próprias coisas dentro das aspas e vai executar o sed com os argumentos -e e s/’/\\’/g. Quando o sed fizer a análise vai substituir as duas contrabarras por uma única e então vai fazer a substituição. Talvez em um algum ponto tenha se pensado que é ótimo todos os comandos Unix usarem o mesmo caractere de escape (escape character). Isto tudo ainda ignora o fato de se ter que colocar contrabarra na frente de contrabarra também, porque as constantes de texto do SQL também estão sujeitas a certas interpretações. Neste caso é melhor preparar o arquivo externamente.

Como os dois-pontos podem aparecer nos comandos, a seguinte regra se aplica: Se a variável não está definida, a seqüência de caracteres “dois-pontos+nome” não é mudada. De qualquer maneira pode-se colocar uma contrabarra antes dos dois-pontos para protegê-lo de interpretação (A sintaxe de dois-pontos para variáveis é o padrão SQL para as linguagens embutidas, como o ecpg. A sintaxe de dois-pontos para “array slices” e “transformações de tipo” são extensões do PostgreSQL, daí o conflito).

Prompt

Os prompts mostrados pelo psql podem ser personalizados conforme a preferência. As três variáveis PROMPT1, PROMPT2 e PROMPT3 contêm cadeias de caracteres e seqüências especiais de escape que descrevem a aparência do prompt. O prompt 1 é o prompt normal que é mostrado quando o psql requisita um novo comando. O prompt 2 é mostrado quando mais entrada é aguardada durante a entrada do comando, porque o comando não foi terminado por um ponto-e-vírgula, ou um apóstrofo não foi fechado. O prompt 3 é mostrado quando se executa um comando SQL COPY e espera-se que as tuplas sejam digitadas no terminal.

O valor da respectiva variável de `prompt` é exibido literalmente, exceto quando um sinal de percentagem (“%”) é encontrado. Dependendo do caractere seguinte, certos outros textos são substituídos. As substituições definidas são:

`%M`

O nome completo do hospedeiro do servidor de banco de dados (com o nome do domínio), ou `[local]` se a conexão for através de um soquete do domínio Unix, ou ainda `[local:`*`/dir/name`*`]`, se o soquete do domínio Unix não estiver no local padrão da compilação.

`%m`

O nome do hospedeiro do servidor de banco de dados, truncado após o primeiro ponto, ou `[local]` se a conexão for através de um soquete do domínio Unix.

`%>`

O número da porta na qual o servidor de banco de dados está na espera.

`%n`

O nome do usuário que se está conectado (não o nome do usuário do sistema operacional local).

`%/`

O nome do banco de dados corrente.

`%~`

Como `%/`, mas a saída será “~” (til) se o banco de dados for o banco de dados padrão.

`%#`

Se o usuário corrente for um superusuário do banco de dados, então “#”, senão “>”.

`%R`

No `prompt` 1 normalmente “=”, mas “^” se estiver no modo linha-única e “!” se a sessão estiver desconectada do banco de dados (o que pode acontecer se o `\connect` falhar). No `prompt` 2 a seqüência é substituída por “-”, “*”, apóstrofo, ou aspas, dependendo se psql aguarda mais entrada devido ao comando não ter terminado ainda, porque está dentro de um comentário `/* ... */`, ou porque está entre apóstrofos ou aspas. No `prompt` 3 a seqüência não se transforma em nada.

`%`*`dígitos`*

Se *`dígitos`* começar por `0x` os caracteres restantes são interpretados como dígitos hexadecimais e o caractere com o código correspondente é substituído. Se o primeiro dígitos for `0` os caracteres são interpretados como um número octal e o caractere correspondente é substituído. Senão um número decimal é assumido.

`%:`*`nome`*`:`

O valor da variável *`nome`* do psql. Veja a seção “*Variáveis*” para obter detalhes.

`%‘`*`comando`*`‘`

A saída do *`comando`*, similar à substituição de “crase” normal.

Para inserir um sinal de percentagem no `prompt` deve-se escrever `%%`. Os prompts padrão são equivalentes a `'%/%R%# '` para os `prompts` 1 e 2, e `'>> '` para o `prompt` 3.

Note: Esta funcionalidade foi plagiada da tcsh.

Miscelânea

O psql retorna 0 se terminar normalmente, 1 se ocorrer um erro fatal próprio (falta de memória, arquivo não encontrado), 2 se a conexão com o servidor teve problema e a sessão não é interativa, e 3 se ocorrer um erro em um script e a variável `ON_ERROR_STOP` estiver definida.

Antes de começar, o psql tenta ler e executar os comandos do arquivo `$HOME/.psqlrc`. Este pode ser usado para configurar o cliente ou o servidor conforme se deseje (usando os comandos `\set` e `SET`).

Biblioteca de histórico GNU

O psql suporta as bibliotecas de histórico e de leitura de linha por ser conveniente para a edição e recuperação da linha. O histórico dos comandos é armazenado no arquivo chamado `.psql_history` no diretório `home`, sendo recarregado quando o psql inicia. Completação por tabulação também é suportado, embora a lógica da completação não pretenda ser a de um analisador SQL. Quando disponíveis, o psql é construído para usar automaticamente estas funcionalidades. Se por algum motivo não se gostar da completação por tabulação, pode-se desativá-la informando isto no arquivo chamado `.inputrc` no diretório `home`:

```
$if psql
set disable-completion on
$endif
```

(Esta não é uma funcionalidade do psql, mas sim do readline. Leia a sua documentação para obter mais detalhes).

Se a biblioteca do readline estiver instalada, mas o psql parecer não usá-la, deve-se ter certeza que o script `configure` do PostgreSQL pode encontrá-la. O script `configure` precisa encontrar a biblioteca `libreadline.a` (ou uma biblioteca compartilhada equivalente) *e* os arquivos de cabeçalho `readline.h` e `history.h` (ou `readline/readline.h` e `readline/history.h`) nos diretórios apropriados. Se os arquivos de biblioteca e de cabeçalhos estiverem instalados em um lugar obscuro, este local deve ser informado ao `configure` como, por exemplo:

```
$ ./configure --with-includes=/opt/gnu/include --with-libs=/opt/gnu/lib  ...
```

Em seguida é necessáro recompilar o psql (não necessariamente toda a árvore do código).

A biblioteca GNU readline pode ser obtida do projeto GNU no servidor de FTP ftp://ftp.gnu.org.

Exemplos

Note: Esta seção somente mostra uns poucos exemplos específicos para o psql. Se for desejado aprender o SQL ou se tornar familiar com o PostgreSQL, é aconselhada a leitura do tutorial que está incluído na distribuição.

O primeiro exemplo mostra como distribuir um comando por várias linhas de entrada. Observe a mudança do `prompt`:

```
testdb=> CREATE TABLE minha_tabela (
testdb(>  first integer not null default 0,
testdb(>  second text
testdb-> );
CREATE
```

Agora veja novamente a definição da tabela:

```
testdb=> \d minha_tabela
             Table "minha_tabela"
 Attribute |  Type   |      Modifier
-----------+---------+--------------------
 first     | integer | not null default 0
 second    | text    |
```

Neste ponto pode-se desejar mudar o `prompt` para algo mais interessante:

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

Vamos assumir que a tabela já esteja com dados e queremos vê-los:

```
peter@localhost testdb=> SELECT * FROM minha_tabela;
 first | second
-------+--------
     1 | one
     2 | two
     3 | three
     4 | four
(4 rows)
```

Esta tabela pode ser mostrada de forma diferente usando-se o comando `\pset`:

```
peter@localhost testdb=> \pset border 2
Border style is 2.
peter@localhost testdb=> SELECT * FROM minha_tabela;
+-------+--------+
| first | second |
+-------+--------+
|     1 | one    |
|     2 | two    |
|     3 | three  |
|     4 | four   |
+-------+--------+
(4 rows)

peter@localhost testdb=> \pset border 0
Border style is 0.
peter@localhost testdb=> SELECT * FROM minha_tabela;
```

```
first second
----- ------
    1 one
    2 two
    3 three
    4 four
(4 rows)

peter@localhost testdb=> \pset border 1
Border style is 1.
peter@localhost testdb=> \pset format unaligned
Output format is unaligned.
peter@localhost testdb=> \pset fieldsep ","
Field separator is ",".
peter@localhost testdb=> \pset tuples_only
Showing only tuples.
peter@localhost testdb=> SELECT second, first FROM minha_tabela;
one,1
two,2
three,3
four,4
```

Alternativamente, podem ser usados os comandos curtos:

```
peter@localhost testdb=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> SELECT * FROM minha_tabela;
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

Apêndice

Problemas

- Nas versões iniciais o psql permitia o primeiro argumento começar logo após o comando (letra-única). Por motivo de compatibilidade isto ainda é suportado de alguma forma, que não será explicada em

detalhes aqui porque é desencorajado. Mas, se forem recebidas mensagens estranhas, tenha isso em mente. Por exemplo:

```
testdb=> \foo
Field separator is "oo",
```

o que talvez não seja o esperado.

- O psql somente trabalha adequadamente com servidores da mesma versão. Isto não significa que outras combinações vão falhar inteiramente, mas problemas sutis, e nem-tão-sutis, podem acontecer.
- Pressionar Control-C durante um “copy in” (envio de dados para o servidor) não apresenta o mais ideal dos comportamentos. Se for recebida uma mensagem como “COPY state must be terminated first”, simplesmente deve-se refazer a conexão entrando com `\c - -`.

pgtclsh

Name

`pgtclsh` — `shell` Tcl cliente do PostgreSQL

Synopsis

`pgtclsh` [*nome_do_arquivo* [*argumentos*...]]

Descrição

O `pgtclsh` é uma interface `shell` Tcl estendida com funções de acesso a bancos de dados do PostgreSQL (Essencialmente, é uma `tclsh` com a `libpgtcl` carregada). Como no caso da `shell` Tcl normal, o primeiro argumento da linha de comando é o nome do arquivo de script, e todos os demais argumentos são passados para o script. Se nenhum nome de arquivo for fornecido, a `shell` será interativa.

Uma `shell` Tcl com Tk e funções PostgreSQL está disponível em pgtksh .

Consulte também

pgtksh , *Guia do Programador do PostgreSQL* (descrição da `libpgtcl`) , tclsh

pgtksh

Name

`pgtksh` — `shell` Tcl/Tk cliente do PostgreSQL

Synopsis

`pgtksh` [*nome_do_arquivo* [*argumentos*...]]

Description

O `pgtksh` é uma interface `shell` Tcl/Tk estendida com funções de acesso a bancos de dados do PostgreSQL (Essencialmente, é uma `tksh` com a `libpgtcl` carregada). Como no caso da `shell` Tcl/Tk normal, o primeiro argumento da linha de comando é o nome do arquivo de script, e todos os demais argumentos são passados para o script. As opções especiais podem ser tratadas pelas bibliotecas do sistema X Window em vez da `shell`. Se nenhum nome de arquivo for fornecido, a `shell` será interativa.

Uma `shell` Tcl com funções PostgreSQL está disponível em pgtclsh .

Consulte também

pgtclsh , *Guia do Programador do PostgreSQL* (descrição da `libpgtcl`) , tclsh , wish

vacuumdb

Name

`vacuumdb` — limpa e analisa um banco de dados do PostgreSQL

Synopsis

`vacuumdb` [*opções_de_conexão*...] [[-d] *nome_bd*] [--full | -f] [--verbose | -v] [--analyze | -z] [--table '*tabela* [(*coluna* [,...])]']
`vacuumdb` [*opções_de_conexão*...] [--all | -a] [--full | -f] [--verbose | -v] [--analyze | -z]

Entradas

O vacuumdb aceita os seguintes argumentos de linha de comando:

-d *nome_bd*
--dbname *nome_bd*

Especifica o nome do banco de dados a ser limpo ou analisado.

-a
--all

Limpa/analisa todos os bancos de dados.

-f
--full

Executa a limpeza completa (“full”).

-v
--verbose

Exibe informações detalhadas durante o processamento.

-z
--analyze

Calcula estatísticas a serem utilizadas pelo otimizador.

-t *tabela* [(*coluna* [,...])]
--table *tabela* [(*coluna* [,...])]

Limpa ou analisa somente a *tabela*. Os nomes das colunas só podem ser especificados juntamente com a opção `--analyze`.

> **Tip:** Se forem especificadas as colunas a serem analisadas, provavelmente será necessário fazer o escape dos parênteses para a `shell`.

O vacuumdb também aceita os seguintes argumentos de linha de comando para os parâmetros de conexão:

-h *hospedeiro*
--host *hospedeiro*

Especifica o nome da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/), é considerado como sendo o diretório do soquete do domínio Unix.

-p *porta*
--port *porta*

Especifica a porta Internet TCP/IP, ou o soquete do domínio local Unix, onde o servidor está aguardando as conexões.

-U *nome_do_usuário*
--username *nome_do_usuário*

Nome do usuário para se conectar.

-W
--password

Força a solicitação da senha.

-e
--echo

Exibe os comandos que o vacuumdb gera e envia para o servidor.

-q
--quiet

Não exibe a resposta.

Saídas

`VACUUM`

O comando foi executado com sucesso.

`vacuumdb: Vacuum failed.`

Aconteceu algum erro. O vacuumdb é apenas um script envoltório. Consulte o comando *VACUUM* e o aplicativo psql para ver uma discussão detalhada das mensagens de erro e dos problemas possíveis.

Descrição

O vacuumdb é um utilitário para fazer a limpeza de bancos de dados do PostgreSQL. O vacuumdb também gera estatísticas internas usadas pelo otimizador de consultas do PostgreSQL.

O vacuumdb é um script envoltório que usa o comando do servidor *VACUUM* através do terminal interativo do PostgreSQL psql. Não existe diferença efetiva entre limpar o banco de dados desta ou daquela maneira. O psql deve ser encontrado pelo script, e o servidor de banco de dados deve estar executando na máquina de destino. Também se aplicam os padrões definidos e as variáveis de ambiente disponíveis para o psql e para a biblioteca cliente libpq.

Utilização

Para limpar o banco de dados `teste`:

```
$ vacuumdb teste
```

Para limpar e analisar para o otimizador o banco de dados chamado `grande_bd`:

```
$ vacuumdb --analyze grande_bd
```

Para limpar uma única tabela chamada `foo` em um banco de dados chamado `xyzzy` e analisar uma única coluna da tabela chamada `bar` para o otimizador:

```
$ vacuumdb --analyze --verbose --table 'foo(bar)' xyzzy
```

III. Aplicativos para o servidor do PostgreSQL

Esta parte contém informações de referência para os aplicativos do servidor e utilitários de suporte do PostgreSQL. Estes comandos só são úteis quando executados no computador onde o servidor de banco de dados está instalado. Outros programas utilitários estão listados em Reference II, *Aplicativos para a estação cliente do PostgreSQL*.

initdb

Name

`initdb` — cria um novo agrupamento de bancos de dados do PostgreSQL

Synopsis

`initdb` --pgdata | -D *diretório* [--username | -U *nome_do_usuário*] [--pwprompt | -W] [--encoding | -E *codificação*] [-L *diretório*] [--noclean | -n] [--debug | -d]

Descrição

O `initdb` cria um novo agrupamento de bancos de dados do PostgreSQL (ou sistema de bancos de dados). Um agrupamento de bancos de dados é uma coleção de bancos de dados que são gerenciados por uma única instância do servidor.

Criar um sistema de banco de dados consiste em criar os diretórios onde os dados dos bancos de dados vão residir, gerar as tabelas do catálogo compartilhado (tabelas que pertencem ao agrupamento como um todo e não a algum banco de dados em particular) e criar o banco de dados `template1`. Quando um novo banco de dados é criado, tudo que existe no banco de dados `template1` é copiado. Este banco de dados contém tabelas do catálogo preenchidas com dados como os tipos de dado primitivos.

O `initdb` deve ser executado pelo mesmo usuário do processo servidor, porque o servidor necessita ter acesso aos arquivos e diretórios criados pelo `initdb`. Como o servidor não deve ser executado como `root`, também não se deve executar o `initdb` como `root` (Na verdade, este vai se recusar a fazê-lo).

Embora o `initdb` tente criar o diretório de dados especificado, muitas vezes não terá permissão para fazê-lo, porque este diretório geralmente está sob um diretório que pertence ao `root`. Para resolver uma situação como esta, deve-se criar um diretório de dados vazio como `root`, depois usar o comando `chown` para mudar o dono deste diretório para o usuário do banco de dados, em seguida executar `su` para se tornar o usuário do banco de dados e, finalmente, executar `initdb` como o usuário do banco de dados.

Opções

--pgdata=*diretório*
-D *diretório*

Esta opção especifica o diretório onde o sistema de banco de dados deve ser armazenado. Esta é a única informação requerida pelo `initdb`, mas pode ser evitado escrevê-la definindo a variável de ambiente `PGDATA`, o que é conveniente porque, depois, o servidor de banco de dados (`postmaster`) poderá localizar o diretório dos bancos de dados usando a mesma variável.

--username=*nome_do_usuário*
-U *nome_do_usuário*

Especifica o nome do superusuário do banco de dados. Por padrão o nome do usuário efetivo executando o `initdb`. Não importa realmente qual é o nome do superusuário, mas deve-se preferir manter o nome costumeiro "postgres", mesmo que o nome do usuário do sistema operacional seja diferente.

--pwprompt
-W

Faz o `initdb` solicitar a senha a ser dada ao superusuário do banco de dados. Se não se pretende usar autenticação por senha, isto não tem importância. Senão, não será possível usar autenticação por senha enquanto não houver uma senha definida.

--encoding=*codificação*
-E *codificação*

Seleciona a codificação do banco de dados de gabarito. Será também a codificação padrão para todos os bancos de dados que vierem a ser criados, a menos que seja substituída por uma outra. Para que a funcionalidade de codificação possa ser usada, esta deve ter sido habilitada em tempo de compilação, quando também pode ser selecionado o valor padrão para esta opção.

Outros parâmetros, menos utilizados, também estão disponíveis:

-L *diretório*

Especifica onde o `initdb` deve encontrar os seus arquivos de entrada para inicializar o sistema de banco de dados. Normalmente não é necessário. Se for necessário especificar a localização será informado explicitamente.

--noclean
-n

Por padrão, quando o `initdb` determina que um erro impediu a criação completa do sistema de bancos de dados, são removidos todos os arquivos porventura criados antes de ser descoberto que não era possível terminar o trabalho. Esta opção impede a limpeza e, portanto, é útil para a depuração.

--debug
-d

Exibe saída de depuração do servidor de `bootstrap` e algumas outras mensagens de menor interesse para o público em geral. O servidor de `bootstrap` é o programa que o `initdb` utiliza para criar as tabelas do catálogo. Esta opção gera uma quantidade tremenda de saída extremamente entediante.

Ambiente

`PGDATA`

Especifica o diretório onde o sistema de bancos de dados deve ser armazenado; pode ser substituído usando a opção `-D`.

Consulte também

postgres, postmaster, *Guia do Administrador do PostgreSQL*

initlocation

Name

`initlocation` — cria uma área secundária de armazenamento de bancos de dados do PostgreSQL

Synopsis

`initlocation` *`diretório`*

Descrição

O initlocation cria uma nova área secundária de armazenamento de bancos de dados do PostgreSQL. Veja em *CREATE DATABASE* a discussão sobre como usar e gerenciar áreas de armazenamento secundárias. Se o argumento não contiver uma barra (/), e não for um caminho válido, será assumido como sendo uma variável de ambiente, a qual é referenciada. Veja os exemplos no final.

Para poder usar este comando é necessário estar autenticado no sistema operacional como o superusuário do banco de dados (usando o comando `su`, por exemplo).

Utilização

Para criar um banco de dados em um local alternativo, usando uma variável de ambiente:

```
$ export PGDATA2=/opt/postgres/data
```

Deve-se parar e reiniciar o `postmaster` para que este enxergue a variável de ambiente `PGDATA2`. O sistema deve ser configurado de maneira que o `postmaster` enxergue `PGDATA2` toda vez que iniciar. Finalmente:

```
$ initlocation PGDATA2
$ createdb -D PGDATA2 testdb
```

Quando os caminhos absolutos são permitidos é possível escrever:

```
$ initlocation /opt/postgres/data
$ createdb -D /opt/postgres/data/testdb testdb
```

ipcclean

Name

`ipcclean` — remove a memória compartilhada e os semáforos de um servidor PostgreSQL abortado

Synopsis

```
ipcclean
```

Descrição

O `ipcclean` remove todos os segmentos de memória compartilhada e os semáforos definidos, pertencentes ao usuário corrente. Foi desenvolvido para ser usado para fazer a limpeza após a queda do servidor PostgreSQL (postmaster). Observe que reiniciar o servidor imediatamente também limpa a memória compartilhada e os semáforos, portanto este comando possui pouca utilidade prática.

Somente o administrador do banco de dados deve executar este programa, porque pode ocasionar um comportamento bizarro (i.e., quedas) se for executado durante uma sessão multiusuária. Se este comando for executado enquanto o postmaster estiver executando, a memória compartilhada e os semáforos alocados pelo postmaster serão eliminados, causando uma falha geral nos processos servidores iniciados pelo postmaster.

Notas

Este script é um “hack” mas, durante estes muitos anos desde que foi escrito, ninguém conseguiu desenvolver uma solução igualmente efetiva e portável. Como agora o postmaster pode se autolimpar, não é provável que o `ipcclean` seja melhorado no futuro.

Este script faz pressuposições sobre o formato da saída do utilitário ipcs que podem não ser verdadeira entre sistemas operacionais diferentes. Portanto, pode ser que não funcione no seu sistema operacional.

pg_ctl

Name

`pg_ctl` — inicia, pára ou reinicia o servidor PostgreSQL

Synopsis

`pg_ctl` start [-w] [-s] [-D *diretório_de_dados*] [-l *nome_do_arquivo*] [-o *opções*] [-p *caminho*]
`pg_ctl` stop [-W] [-s] [-D *diretório_de_dados*] [-m s[mart] | f[ast] | i[mmediate]]
`pg_ctl` restart [-w] [-s] [-D *diretório_de_dados*] [-m s[mart] | f[ast] | i[mmediate]] [-o *opções*]
`pg_ctl` reload [-s] [-D *diretório_de_dados*]
`pg_ctl` status [-D *diretório_de_dados*]

Descrição

O pg_ctl é um utilitário para iniciar, parar ou reiniciar o postmaster, o servidor PostgreSQL, ou exibir o status de um postmaster ativo. Embora o postmaster possa ser iniciado manualmente, o pg_ctl encapsula tarefas como redirecionar a saída do `log`, desconectar do terminal e do grupo de processo de maneira adequada, além de fornecer opções convenientes para uma parada controlada.

No modo iniciar (`start`), um novo postmaster é lançado. O servidor inicia em segundo plano, e a entrada padrão lê de `/dev/null`. A saída padrão e o erro padrão são ambos adicionados a um arquivo de `log`, se a opção `-l` for usada, ou são redirecionados para a saída padrão do pg_ctl (não o erro padrão). Se o arquivo de `log` não for definido, a saída padrão do pg_ctl deve ser redirecionada para um arquivo ou ser enviada para outro processo (através de um `pipe`) como, por exemplo, para um programa para rotação de `logs`, senão o postmaster vai escrever sua saída no terminal de controle (do segundo plano) e não vai se desconectar do grupo de processo da `shell`.

No modo parar (`stop`), o postmaster que está executando no diretório de dados especificado é parado. Três métodos diferentes de parada podem ser selecionados pela opção `-m`: O modo “Smart” (inteligente) aguarda todos os clientes se desconectarem. Este é o modo padrão. O modo “Fast” (rápido) não aguarda os clientes se desconectarem. Todas as transações ativas são desfeitas (`rollback`), os clientes são desconectados à força e, em seguida, o servidor é parado. O modo “Immediate” (imediato) aborta todos os processor servidores sem executar uma parada limpa, obrigando um processamento de recuperação ao reiniciar.

O modo reiniciar (`restart`) executa uma parada seguida por um início. Permite mudar as opções de linha de comando do postmaster.

O modo recarregar (`reload`) simplesmente envia o sinal SIGHUP para o postmaster, fazendo com que este releia os arquivos de configuração (`postgresql.conf`, `pg_hba.conf`, etc.). Permite mudar as opções do arquivo de configuração que não requerem um reinício completo para ter efeito.

O modo `status` verifica se o postmaster está executando e, se estiver, exibe o PID e as opções de linha de comando que foram usadas para chamá-lo.

Opções

-D `diretório_de_dados`

Especifica a localização dos arquivos de banco de dados. Se for omitido, a variável de ambiente `PGDATA` é usada.

-l `nome_do_arquivo`

Apensa a saída do `log` do servidor ao *nome_do_arquivo*. Se o arquivo não existir é criado. A `umask` é definida como 077, não permitindo o acesso ao arquivo de `log` pelos outros usuários por padrão.

-m `modo`

Especifica o modo de parar (`shutdown`). O *modo* pode ser `smart`, `fast` ou `immediate`, ou a primeira letra de um desses três.

-o `opções`

Especifica opções a serem passadas diretamente para o postmaster.

Os parâmetros são geralmente envoltos por aspas(`"`) ou apóstrofos (`'`) para garantir que são passados como um grupo.

-p `caminho`

Especifica a localização do arquivo executável `postmaster`. Por padrão o `postmaster` é pego do mesmo diretório do pg_ctl ou, se falhar, do diretório de instalação. Não é necessário usar esta opção a menos que esteja se fazendo algo diferente do usual e recebendo uma mensagem de erro informando que o `postmaster` não foi encontrado.

-s

Mostra somente os erros, sem nenhuma mensagem informativa.

-w

Aguarda o início ou a parada terminar. Espera no máximo 60 segundos. Este é o padrão para a parada.

-W

Não aguarda o início ou a parada terminar. Este é o padrão para o início e o reínicio.

Arquivos

Se o arquivo `postmaster.opts.default` existir no diretório de dados, o conteúdo deste arquivo será passado como opções para o postmaster, a menos que seja substituído pela opção `-o`.

Exemplos

Iniciar o postmaster

Para iniciar o postmaster:

```
$ pg_ctl start
```

Exemplo de iniciar o postmaster, bloqueando até que o postmaster esteja ativo:

```
$ pg_ctl -w start
```

Para iniciar o postmaster utilizando a porta 5433 e executando sem o `fsync` pode-se usar:

```
$ pg_ctl -o "-F -p 5433" start
```

Parar o postmaster

```
$ pg_ctl stop
```

pára o postmaster. A chave `-m` permite controlar *como* o servidor vai parar.

Reiniciar o postmaster

Praticamente equivale a parar o postmaster e iniciá-lo novamente, exceto que o pg_ctl salva e reutiliza as opções de linha de comando que foram passadas para a instância executando anteriormente. Para reiniciar o postmaster da forma mais simples possível:

```
$ pg_ctl restart
```

Para reiniciar o postmaster, aguardando o término da parada e da inicialização:

```
$ pg_ctl -w restart
```

Para reiniciar usando a porta 5433 e desativando o fsync após o reinício:

```
$ pg_ctl -o "-F -p 5433" restart
```

Exibir o status do postmaster

Abaixo segue uma um exemplo da saída de status mostrada pelo pg_ctl:

```
    $ pg_ctl status
    pg_ctl: postmaster is running (pid: 13718)
Command line was:
/usr/local/pgsql/bin/postmaster ’-D’ ’/usr/local/pgsql/data’ ’-p’ ’5433’ ’-B’ ’128’
```

Esta é a linha de comandos que seria usada no modo de reinício.

Bugs

Aguardar o término do início não é uma operação bem definida, podendo falhar se o controle de acesso for definido de uma maneira que o cliente não possa se conectar sem intervenção manual. Portanto, deve ser evitado.

Consulte também

postmaster, *Guia do Administrador do PostgreSQL*

pg_passwd

Name

`pg_passwd` — muda um arquivo secundário de senhas do PostgreSQL

Synopsis

```
pg_passwd nome_do_arquivo
```

Descrição

O pg_passwd é uma ferramenta para tratar arquivos de senhas texto. Estes arquivos podem controlar a autenticação dos clientes do servidor PostgreSQL. Mais informações sobre como configurar este mecanismo de autenticação podem ser encontradas no *Guia do Administrador*.

O formato do arquivo de senhas texto é uma entrada por linha; os campos de cada entrada são separados por dois-pontos (:). O primeiro campo é o nome do usuário, o segundo campo é a senha criptografada. Demais campos são ignorados (para permitir que arquivos de senha sejam compartilhados entre aplicativos que usam formato similar). O pg_passwd permite aos usuários adicionar entradas neste arquivo interativamente, alterar as senhas das entradas existentes, e criptografar estas senhas.

Forneça o nome do arquivo de senhas como um argumento para o pg_passwd. Para ser usado pelo PostgreSQL, o arquivo deve estar localizado no diretório de dados do servidor, e o nome base do arquivo deve estar especificado no arquivo de controle de acesso `pg_hba.conf` .

```
$ pg_passwd /usr/local/pgsql/data/senhas
File "/usr/local/pgsql/data/senhas" does not exist.  Create? (y/n): y
Username: guest
Password:
Re-enter password:
```

onde as mensagens `Password:` e `Re-enter password:` requerem a entrada da mesma senha, que não é exibida no terminal. Observe que a senha está limitada a oito caracteres úteis por restrições da rotina de criptografia(3) da biblioteca padrão.

O arquivo original de senhas é renomeado para `senhas.bk`.

Para usar este arquivo de senhas, coloque uma linha como a mostrada abaixo no arquivo `pg_hba.conf`:

```
host  meu_bd     133.65.96.250   255.255.255.255 password senhas
```

que permite o acesso ao banco de dados `meu_bd` pelo hospedeiro `133.65.96.250` usando as senhas listadas no arquivo `senhas` (e somente aos usuários listados neste arquivo).

> **Note:** Também é útil existirem entradas com o campo de senha vazio, o que é diferente de uma senha vazia, no arquivo de senhas. Estas entradas permitem restringir os usuários que podem acessar o sistema. Estas entradas não podem ser gerenciadas pelo pg_passwd, mas o arquivo de senhas pode ser editado manualmente.

Consulte também

Guia do Administrador do PostgreSQL

postgres

Name

`postgres` — executa o servidor PostgreSQL no modo monousuário

Synopsis

`postgres` [-A 0 | 1] [-B *num_buffers*] [-c *nome*=*valor*] [-d *nível_de_depuração*] [-D *diretório_de_dados*] [-e] [-E] [-f s | i | t | n | m | h] [-F] [-i] [-N] [-o *nome_do_arquivo*] [-O] [-P] [-s | -t pa | pl | ex] [-S *memória_para_ordenação*] [-W *segundos*] [--*nome*=*valor*] *banco_de_dados*
`postgres` [-A 0 | 1] [-B *num_buffers*] [-c *nome*=*valor*] [-d *nível_de_depuração*] [-D *diretório_de_dados*] [-e] [-f s | i | t | n | m | h] [-F] [-i] [-o *nome_do_arquivo*] [-O] [-p *banco_de_dados*] [-P] [-s | -t pa | pl | ex] [-S *memória_para_ordenação*] [-v *versão_do_protocolo*] [-W *segundos*] [--*nome*=*valor*]

Descrição

O executável `postgres` é o verdadeiro processo servidor que processa os comandos do PostgreSQL. Normalmente não é chamado diretamente; em vez dele um servidor multiusuário postmaster é ativado.

A segunda forma mostrada acima é como o postgres é chamado pelo postmaster (somente conceitualmente, porque o `postmaster` e o `postgres` são, de fato, o mesmo programa); não deve ser chamado diretamente desta maneira. A primeira forma chama o servidor diretamente em modo monousuário interativo. A utilização mais freqüente deste modo é durante a inicialização pelo initdb. Algumas vezes é utilizado para depuração ou para recuperação de desastre.

Quando chamado em modo interativo a partir da `shell`, o usuário pode entrar com comandos e os resultados são exibidos na tela, mas em uma forma que é mais útil para os desenvolvedores do que para os usuários finais. Deve-se notar que executar o servidor em modo monousuário não é muito apropriado para a depuração do servidor, porque não vai acontecer nenhuma comunicação entre processos e bloqueio genuínos.

Ao executar o servidor autônomo, o usuário da sessão será definido como o usuário com o identificador 1. Este usuário não necessita existir, portanto o servidor autônomo pode ser usado para recuperar manualmente certos tipos de danos acidentais dos catálogos do sistema. Poderes implícitos de superusuário são concedidos ao usuário com identificador igual a 1 no modo autônomo.

Opções

Quando o postgres é iniciado pelo postmaster, herda todas as opções definidas para este. Adicionalmente, opções específicas do postgres podem ser passadas pelo postmaster usando a chave `-o`.

Pode-se evitar digitar a maior parte destas opções usando um arquivo de configuração. Consulte o *Guia do Administrador* para obter detalhes. Algumas opções (seguras) também podem ser definidas pelo cliente ao se conectar, de um modo dependente do aplicativo. Por exemplo, se a variável de ambiente `PGOPTIONS`

estiver definida, então os clientes baseados na `libpq` passam esta cadeia de caracteres para o servidor, que vai interpretá-la como uma opção de linha de comando do postgres.

Finalidade geral

As opções `-A`, `-B`, `-c`, `-d`, `-D`, `-F` e `--name` possuem o mesmo significado que têm para o postmaster.

-e

Define o estilo padrão da data como “European”, indicando que a regra “dia precedendo o mês” (em vez de mês precedendo o dia) deve ser usada para interpretar a entrada de datas ambíguas, e que o dia precede o mês em certos formatos de exibição de data. Consulte o *Guia do Usuário do PostgreSQL* para obter mais informações.

-o *nome_do_arquivo*

Envia toda a saída de depuração e de erros para o *nome_do_arquivo*. Se o processo servidor estiver executando sob o postmaster então esta opção será ignorada, e a `stderr` herdada do postmaster será usada.

-P

Ignora os índices do sistema ao varrer/atualizar as tuplas do sistema. O comando `REINDEX` para as tabelas/índices do sistema requer que esta opção seja utilizada.

-s

Exibe a informação de tempo e outras estatísticas ao final de cada comando. Útil para efetuar medições ou para definir o número de `buffers`.

-S *memória_para_ordenação*

Especifica a quantidade de memória a ser usada pelas ordenações e `hashes` internos antes de recorrer a arquivos temporários em disco. O valor é especificado em `kilobytes`, sendo o valor padrão 512 `kilobytes`. Observe que para uma consulta complexa, diversas ordenações e/ou `hashes` podem ser executados em paralelo, podendo cada um utilizar o tanto de *memória_para_ordenação* kilobytes antes de começar a escrever os dados em arquivos temporários.

Opções para o modo autônomo

banco_de_dados

Especifica o nome do banco de dados a ser acessado. Se for omitido o padrão é o nome do usuário.

-E

Exibe todos os comandos.

-N

Desativa o uso do caractere de nova-linha como delimitador do comando.

Opções semi-internas

Existem várias outras opções que podem ser especificadas, usadas principalmente para fins de depuração, que estão mostradas aqui somente para o uso pelos desenvolvedores de sistema do PostgreSQL. *A utilização de qualquer uma destas opções é altamente desaconselhada.* Além disso, qualquer uma destas opções poderá mudar ou desaparecer em versões futuras sem nenhum aviso.

-f { s | i | m | n | h }

Proíbe o uso de um método de varredura ou de junção em particular: `s` e `i` desativam as varreduras seqüenciais e de índices respectivamente, enquanto que `n`, `m` e `h` desativam as junções de laço-aninhado, mesclagem e `hash` respectivamente.

> **Note:** Nem as varreduras seqüenciais nem as junções de laço aninhado podem ser desativadas completamente; as opções `-fs` e `-fn` simplesmente desencorajam o otimizador de usar estes tipos de plano se houver alguma outra alternativa.

-i

Proíbe a execução da consulta, mas mostra a árvore do plano.

-O

Permite modificar a estrutura das tabelas do sistema. Usado pelo initdb.

-p *banco_de_dados*

Indica que este servidor foi inicializado pelo postmaster fazendo suposições diferentes sobre o gerenciamento do `buffer pool`, descritores de arquivos, etc...

-t pa[rser] | pl[anner] | e[xecutor]

Exibe estatísticas de tempo para cada comando, relacionando com cada um dos principais módulos do sistema. Esta opção não pode ser usada junto com a opção `-s`.

-v *versão_do_protocolo*

Especifica o número da versão do protocolo cliente/servidor a ser usado por esta sessão em particular.

-W *segundos*

Tão logo esta opção seja encontrada, o processo adormece a quantidade especificada de segundos, dando ao desenvolvedor tempo para ligar o depurador ao processo servidor.

Utilização

Inicie o servidor autônomo com um comando do tipo:

```
postgres -D $PGDATA outras_opções meu_bd
```

Forneça o caminho correto para a área de banco de dados com `-D`, ou garanta que a variável de ambiente `PGDATA` esteja definida. Também especifique o nome do banco de dados em que deseja trabalhar.

Normalmente, o servidor autônomo trata o caractere de nova-linha como término da entrada; não existe inteligência com relação ao ponto-e-vírgula, como existe no psql. Para continuar um comando por várias linhas, deve-se digitar uma contrabarra (\) logo antes de cada nova-linha, exceto a última.

Mas se for usada a chave de linha de comando `-N`, o caractere de nova-linha não termina a entrada do comando. O servidor vai ler a entrada padrão até a marca de fim-de-arquivo (EOF) e, então, vai processar a entrada como sendo a cadeia de caracteres de um único comando. A seqüência contrabarra nova-linha não recebe tratamento especial neste caso.

Para sair da sessão tecle EOF (geralmente **Control**+**D**). Se for usado o `-N`, então dois EOFs consecutivos são necessários para sair.

Observe que o servidor autônomo não fornece funcionalidades sofisticadas para edição de linha (não existe o histórico dos comandos, por exemplo).

Consulte também

initdb, ipcclean, postmaster

postmaster

Name

`postmaster` — servidor de banco de dados multiusuário do PostgreSQL

Synopsis

`postmaster` [-A 0 | 1] [-B *num_buffers*] [-c *nome*=*valor*] [-d *nível_de_depuração*] [-D *diretório_de_dados*] [-F] [-h *nome_do_hospedeiro*] [-i] [-k *diretório*] [-l] [-N *num_max_conexões*] [-o *opções_extras*] [-p *porta*] [-S] [--*nome*=*valor*] [-n | -s]

Descrição

O postmaster é o servidor de banco de dados multiusuário do PostgreSQL. Para um aplicativo cliente acessar um banco de dados deve se conectar (através de uma rede ou localmente) a um postmaster. O postmaster então inicia um processo servidor separado (“postgres”) para manter a conexão. O postmaster também gerencia a comunicação entre os processos servidores.

Por padrão, o postmaster inicia em primeiro plano (foreground) e envia as mensagens de `log` para a saída padrão. Na prática o postmaster deve ser iniciado como um processo em segundo plano (background), provavelmente durante a inicialização do sistema operacional.

Um postmaster gerencia sempre os dados de exatamente um agrupamento de bancos de dados. Um agrupamento de bancos de dados é uma coleção de bancos de dados que é armazenada em um local comum no sistema de arquivos. Quando o postmaster inicia necessita saber a localização dos arquivos do agrupamento de bancos de dados (“área de dados”), o que é feito através da opção de chamada `-D`, ou através da variável de ambiente `PGDATA`; não existe nenhum valor padrão. Mais de um processo postmaster pode estar executando no sistema operacional no mesmo instante, desde que utilizem áreas de dados diferentes e portas de comunicação diferentes (veja abaixo). A área de dados é criada pelo initdb.

Opções

Consulte o *Guia do Administrador* para ver uma discussão detalhada das opções. Pode-se evitar digitar a maior parte destas opções usando um arquivo de configuração. O postmaster aceita os seguintes argumentos de linha de comando:

-A 0|1

Ativa a verificação das assertivas de tempo de execução, o que é uma ajuda de depuração para detectar enganos de programação. Só está disponível quando é habilitada durante a compilação. Se for, o padrão é ativa.

-B *num_buffers*

Define o número de `buffers` compartilhados para uso pelos processos servidores (SHARED_BUFFERS = num_buffers). Por padrão 64 `buffers`, cada um de 8 kB.

-c `nome=valor`

Define o parâmetro de tempo de execução designado. Consulte o *Guia do Administrador* para obter a relação destes parâmetros e as suas descrições. A maior parte das outras opções de linha de comando são, na verdade, formas curtas de atribuição destes parâmetros. A opção `-c` pode aparecer várias vezes para definir vários parâmetros.

-d *`nível_de_depuração`*

Define o nível de depuração (DEBUG_LEVEL = nível_de_depuração). Quanto maior for definido este valor, mais saída de depuração será escrita no log do servidor. O padrão é 0, que significa sem depuração. Valores até 4 são úteis; números maiores não produzem nenhuma saída adicional.

-D *`diretório_de_dados`*

Especifica a localização do diretório de dados no sistema de arquivos. Veja a discussão acima.

-F

Desativa as chamadas a `fsync` para melhorar o desempenho, correndo o risco de corrupção dos dados na ocorrência de uma falha do sistema (FSYNC = OFF). Leia a documentação antes de usar esta opção!

-h *`nome_do_hospedeiro`*

Especifica o nome ou o endereço do hospedeiro TCP/IP no qual o postmaster vai aguardar as conexões dos aplicativos clientes (VIRTUAL_HOST = nome_do_hospedeiro). Por padrão aguarda em todos os endereços configurados (incluindo localhost).

-i

Permite os clientes se conectarem via TCP/IP (Domínio da Internet) (TCPIP_SOCKET = TRUE). Sem esta opção, somente as conexões via soquete do domínio local Unix são aceitas.

-k *`diretório`*

Especifica o diretório do soquete do domínio Unix, no qual o postmaster está aguardando as conexões dos aplicativos clientes (UNIX_SOCKET_DIRECTORY = diretório). Normalmente o padrão é `/tmp`, mas pode ser mudado em tempo de compilação.

-l

Ativa as conexões seguras usando SSL (SSL = TRUE). A opção `-i` também é requerida. Deve ter sido compilado com SSL habilitado para ser possível o uso desta opção.

-N *`num_max_conexões`*

Define o número máximo de conexões de clientes aceitas por este postmaster (MAX_CONNECTIONS = num_max_conexões). Por padrão este valor é 32, mas pode ser definido tão alto quanto o sistema operacional suportar (Observe que a opção `-B` deve ser pelo menos o dobro da opção `-N`. Veja a discussão sobre os recursos do sistema requeridos para a conexão de um grande número de clientes no *Guia do Administrador*).

-o *`opções_extras`*

As opções no estilo linha de comando especificadas nas *`opções_extras`* são passadas para todos os processos servidores começando por este postmaster. Consulte o postgres para ver as possibilidades. Se a cadeia de caracteres contendo a opção contiver espaços, toda a cadeia de caracteres deve vir entre apóstrofos (`'`).

-p *porta*

Especifica a porta TCP/IP, ou o soquete do domínio local Unix, onde o postmaster está aguardando as conexões dos aplicativos cliente (PORT = porta). Por padrão o valor da variável de ambiente `PGPORT` ou, se `PGPORT` não estiver definida, o valor estabelecido durante a compilação (normalmente 5432). Se for especificada outra porta diferente da porta padrão, então todos os aplicativos cliente devem especificar a mesma porta usando a opção de linha de comando ou a variável de ambiente `PGPORT`.

-S

Especifica que o processo postmaster deve iniciar no modo silencioso, ou seja, será dissociado do terminal do usuário, iniciará seu próprio grupo de processos e redirecionará sua saída padrão e erro padrão para `/dev/null`.

O uso desta chave descarta toda a saída para o `log`, o que provavelmente não é o desejado, porque torna muito difícil a solução dos problemas. Veja abaixo uma maneira mais adequada de iniciar o postmaster em segundo plano.

--*nome*=*valor*

Define o parâmetro de tempo de execução designado; uma forma mais curta da opção `-c`.

Duas opções adicionais de linha de comando estão disponíveis para a depuração de problemas que fazem o servidor terminar anormalmente. Estas opções controlam o comportamento do postmaster nesta situação, e *nenhuma delas foi feita para ser usada durante a operação normal*.

A estratégia comum para esta situação é notificar a todos os outros servidores que eles devem terminar e, então, reinicializar a memória compartilhada e os semáforos. Isto é necessário porque um servidor errante pode ter corrompido algum estado compartilhado antes de terminar.

Estas opções caso-especial são:

-n

O postmaster não irá reinicializar as estruturas de dado compartilhadas. Um programador de sistemas com conhecimento adequado poderá, então, usar um depurador para examinar a memória compartilhada e o estado do semáforo.

-s

O postmaster irá parar todos os outros processos servidores enviando o sinal `SIGSTOP`, mas não irá fazê-los terminar, permitindo aos programadores de sistema coletar os "core dumps" de todos os processos servidores manualmente.

Saídas

`semget: No space left on device`

Se esta mensagem for recebida deve-se executar o comando ipcclean e em seguida, tentar iniciar o postmaster novamente. Se ainda assim não funcionar, provavelmente será necessário configurar

o núcleo (kernel) para a memória compartilhada e os semáforos conforme descrito nas notas de instalação. Se forem executadas várias instâncias do postmaster em um único hospedeiro, ou se o núcleo tiver memória compartilhada e/ou limites de semáforo particularmente pequenos, provavelmente será necessário reconfigurar o núcleo para aumentar os parâmetros de memória compartilhada ou de semáforos.

> **Tip:** Pode-se conseguir adiar a reconfiguração do núcleo diminuindo-se `-B` para reduzir o consumo de memória compartilhada do PostgreSQL, e/ou reduzindo-se `-N` para reduzir o consumo de semáforos.

`StreamServerPort: cannot bind to port`

Se esta mensagem for vista, deve-se ter certeza de que não existe nenhum outro processo postmaster executando no mesmo computador usando o mesmo número de porta. A maneira mais fácil de se ver é usando o comando

```
$ ps ax | grep postmaster
```

ou

```
$ ps -e | grep postmaster
```

dependendo do sistema operacional.

Havendo certeza de que nenhum outro processo postmaster está executando, e o erro continuar acontecendo, deve-se tentar especificar uma porta diferente usando a opção `-p`. É possível acontecer este erro se o postmaster for terminado e imediatamente reiniciado usando a mesma porta; neste caso deve-se simplesmente aguardar alguns segundos até que o sistema operacional feche a porta antes de tentar novamente. Finalmente, este erro pode acontecer se for especificado um número de porta que o sistema operacional considera ser reservado. Por exemplo, muitas versões do Unix consideram os números de porta abaixo de 1024 como sendo *trusted* (confiadas) e só permite o acesso aos superusuários do Unix.

Notas

Sempre que for possível *não* use o `SIGKILL` para terminar o postmaster. Isto impede que o postmaster libere os recursos do sistema utilizados (por exemplo, memória compartilhada e semáforos) antes de terminar.

Para terminar o postmaster normalmente, os sinais `SIGTERM`, `SIGINT` ou `SIGQUIT` podem ser usados. O primeiro aguarda todos os clientes terminarem antes de fechar, o segundo força a desconexão de todos os clientes e o terceiro fecha imediatamente sem um `shutdown` apropriado, acarretando a execução da recuperação ao reiniciar.

O comando utilitário pg_ctl pode ser usado para iniciar e terminar o postmaster com segurança e conforto.

As opções `--` não funcionam no FreeBSD nem no OpenBSD. Use o `-c` em seu lugar. Esta é uma falha destes sistemas operacionais; uma versão futura do PostgreSQL disponibilizará uma forma de contornar este problema, caso não seja corrigido.

Utilização

Para iniciar o postmaster em segundo plano usando os valores padrão:

```
$ nohup postmaster >logfile 2>&1 </dev/null &
```

Para iniciar o postmaster usando uma porta específica:

```
$ postmaster -p 1234
```

Este comando inicia o postmaster se comunicando através da porta 1234. Para se conectar a este postmaster usando o psql, deve-se executar:

```
$ psql -p 1234
```

ou definir a variável de ambiente `PGPORT`:

```
$ export PGPORT=1234
$ psql
```

Parâmetros de tempo de execução nomeados podem ser definidos usando-se um destes estilos:

```
$ postmaster -c sort_mem=1234
$ postmaster --sort-mem=1234
```

As duas formas substituem o que estiver definido para `sort_mem` em `postgresql.conf`. Observe que os sublinhados nos nomes dos parâmetros podem ser escritos na linha de comando com o caractere sublinhado ou o traço (`dash`).

> **Tip:** Exceto para experimentos de curta duração, é provavelmente uma prática melhor editar as definições no arquivo `postgresql.conf` do que depender das chaves da linha de comando para definir os parâmetros.